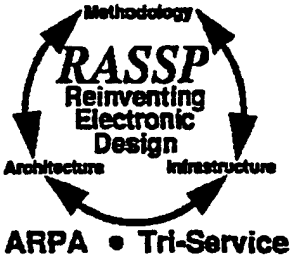
WL-TR-96-1024

ADA/C TO BEHAVIORAL VHDL TRANSLATOR

ROBERT J SHERAGA

JRS RESEARCH LABORATORIES INC
1036 WEST TAFT AVE
ORANGE CA 92665-4121



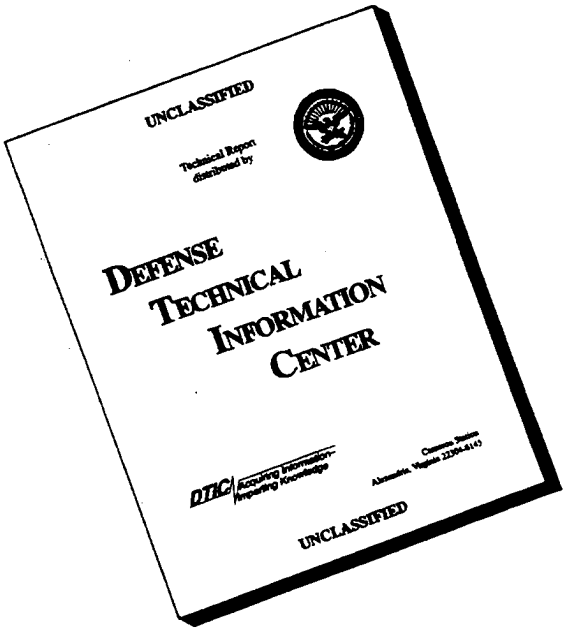
JANUARY 1996

FINAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7623

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

GARY FECHER, Project Engineer
Computer Aided Engineering Tech
System Technology Branch

DAVID A. ZANN, Chief
System Technology Branch
Avionics Directorate

STANLEY E. Wagner, Chief
System Concepts & Simulation Div
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAST-1, WPAFB, OH 45433-7318 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1996	FINAL REPORT

4. TITLE AND SUBTITLE

Ada/C TO BEHAVIORIAL VHDL TRANSLATOR

5. FUNDING NUMBERS

C F33615-94-C-1497
PE 63739E
PR A268
TA 02
WU 17

6. AUTHOR(S)

ROBERT J. SHERAGA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

JRS RESEARCH LABORATORIES INC.
1036 W. TAFT AVE
ORANGE, CA 92665-4121

8. PERFORMING ORGANIZATION REPORT NUMBER

FR.0192-00

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433- 7623

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

WL-TR-96-1024

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION IS UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This is the final report associated with RASSP BAA contract sponsored by Wright Laboratory whose main objective is to develop two source language Translators, an Ada-to-VHDL Translator and a C-to-VHDL Translator. These are referred to collectively as the High-Level Language to VHDL Translators (HVT). The Translators perform source-to-source code translation, i.e., given an input program coded in Ada or C, the output is a behavioral VHDL source coded program which is functionally equivalent to the original input program.

14. SUBJECT TERMS

High-Level-Language to VHDL Translator (HVT), RASSP, VHDL, JRS RASSP Architectural Selection Toolset (AST)

15. NUMBER OF PAGES

80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Contents

Preface

This document is prepared as CDRL 1, Final Report, of US Air Force contract F33615-94-C-1497, called ADA/C TO BEHAVIORAL VHDL TRANSLATOR. The report is organized in the following Sections:

- Section 1 presents the Introduction and Overview of the project
- Section 2 presents the Task Summary that drove the technical approach
- Section 3 presents the Translator Product Descriptions
- Section 4 describes the Ada-to-VHDL Translator
- Section 5 describes the C-to-VHDL Translator
- Section 6 concludes with the Recommendations for Future Work
- Appendix A provides an Ada-to-VHDL Translator example
- Appendix B provides a C-to-VHDL Translator example

Chapter 1

Introduction

1.1 Overview

This is the Final Report associated with a RASSP BAA contract sponsored by Wright Laboratory whose main objective is to develop two source language Translators, an Ada-to-VHDL Translator and a C-to-VHDL Translator. These are referred to collectively as the High-Level Language to VHDL Translators (HVT), or simply the Translators.

This report provides a description of the Translators, the rationale for developing them, a summary of the project tasks, and recommendations for future work in this area.

1.2 Objectives

The objectives of the HVT project are:

- To design and develop source code translators from Ada-to-VHDL and C-to-VHDL.
- To integrate these Translators with the JRS RASSP Architectural Selection Toolset (AST).
- To provide for technology transfer of the Translators both through the RASSP program and as standalone commercial products.

1.3 Scope

The Translators perform source-to-source code translation, i.e., given an input program coded in Ada or C, the output is a behavioral VHDL source code program which is functionally equivalent to the original input program. The input programs must be valid (i.e., syntactically correct) Ada and C programs; the Translators perform some compiler-type checking, but very minimal compared to a conventional language compiler.

Clearly, not every valid Ada or C program can be translated into an equivalent VHDL program, since both languages support features which have no counterpart in VHDL. The Translators are, in general, limited to those features which are directly supportable in VHDL; however, there are some exceptions to this rule. In any case, the Translators accept any valid Ada or ANSI C program as input, and issue warning and/or error messages for any statements which cannot be translated into VHDL.

The primary orientation of the Translators is toward the primitive libraries which are part of the Reusable Software Subsystem (RSS), a subsystem of the AST. Although the Translators are not limited specifically to these programs, neither are they designed or intended to support completely arbitrary Ada and C programs. For example, input programs which interact with an operating system or device, while perhaps being syntactically acceptable for translation, may not represent any meaningful behavior in a VHDL environment. A limited runtime support library is provided for the translated VHDL routines, which includes primarily basic math routines. Specifically, the C-to-VHDL Translator does not support the full standard ANSI C library.

1.4 Purpose

The Translators fulfill a key "missing link" in the Architectural Selection Toolset (AST) for the RASSP program and other IDAS Network Synthesis applications, for the following reason:

1. The architectural selection/network synthesis process is driven by a flow graph representation of a signal processing application.

2. The nodes in the flow graph are reusable software elements from Ada and/or C primitive libraries.

3. The resultant architectural model is created in VHDL.

4. To simulate the architectural model requires behavioral VHDL representations of the library primitives, but currently they only exist in Ada or C.

Thus, developing the Translators, and integrating them into the Network Synthesis environment, enables the system to "close the loop" by automatically translating the primitives into VHDL and running a behavioral simulation of a synthesized network.

As a standalone tool, the Translators provide an automated method for translating reusable code libraries from Ada or C into behaviors in VHDL which can be directly simulated. This provides a generalized tool to aid the process of hardware/software codesign, which can be used either standalone or incorporated into a larger tool framework.

1.5 Rationale

One of the major problems facing the RASSP Program is that of being able to capture, in executable (i.e., simulatable) form a complete representation of a signal processing system early in the design process. Such a representation is of great value in that it would:

- Provide the means for performing comprehensive evaluations of a system design at an early stage, where change is inexpensive.
- Provide the detailed behavioral specifications needed to drive the lower level design processes that ultimately generate the detailed hardware and software solutions.

JRS is providing, to the Martin Marietta Corporation (MMC) RASSP Design System, the Architectural Selection Toolset. This Toolset contains, among other things, network (multiprocessor) synthesis tools that address issues of hardware/software codesign, assignment of behaviors to architectural entities, and simulation at the network (multiprocessor) architecture

level. The architectural model is generated in VHDL and is simulatable in a standard VHDL environment. In the JRS network synthesis tools, functional simulations are performed and functionality is verified, prior to architectural synthesis, in the PGSE Simulation Tools; after synthesis, the simulations generate performance and resource utilization data.

The network architecture level simulation simulates the execution of behaviors on a collection of processing elements (programmable and non-programmable), memories of various types, and communications elements that transfer data between the various memories. The functional behaviors are those that the signal processing system is being designed to execute. They are originally expressed as functional nodes in a data flow graph and have underlying executable Ada (or C) programs that explicitly define the functionality involved. These programs are referred to as Primitives and are provided and maintained in a reusable parts Library as part of the Architectural Selection Toolset (AST).

In the VHDL Simulation of an architecture at the network level, there is currently no actual execution of the functional Primitives and, consequently, the simulation does not deal with real data; rather, it deals with token representations of the data elements as they flow about the architecture. Timelines are generated based on library formulas or rules, as are memory utilization data, contention information, and other resource conflict data of interest. Not actually executing the functional behaviors of the application system prevents the VHDL Simulation from being used for functional testing and verification at the architectural level; the simulation is only useful for Performance and Resource Utilization dynamic analysis. The reason for not actually executing functional behaviors in the architectural level simulation is that the functional behaviors exist only in Ada (or C) form; they do not exist in VHDL.

The Translators provide the means to generate a complete functional VHDL Model at the architectural level for networks. They help satisfy the need for the originally stated RASSP requirement "to capture, in executable (i.e., simulatable) form, a complete representation of a signal processing system early in the design process". They aid in providing a transportable model that can be used by multiple organizations for independent assessment and evaluation of the design captured in the model, against requirements and alternative solutions. They also provide behavioral specifications, that are implementation independent, that may be used as input to lower level design processes for generation of detailed hardware/software solutions.

1.6 Related Documents

The following documents were created as part of this contract, in addition to this Final Report:

1. *Ada/C to Behavioral VHDL Translator Requirements Specification*, JRS Document No. REQ.0172-00, November 22, 1994. Defines the Translator requirements, restrictions and limitations, and an overview of the technical approach.

2. *Ada/C to Behavioral VHDL Translator Software Test Plan*, JRS Document No. REQ.0173-00, December 15, 1994. Defines the requirements and procedures for testing and validating the Translators at the unit, integration, and system test levels.

3. *Ada/C to Behavioral VHDL Translator Software Development Plan*, JRS Document No. REQ.0175-00, December 20, 1994. Defines the tasks, milestones, and schedule for developing the Translators.

4. *Ada/C to Behavioral VHDL Translator Software Demonstration Plan*, JRS Document No. REQ.0176-00, December 22, 1994. Defines the formal demonstrations and procedures to demonstrate that the Translators function correctly and satisfy their requirements.

5. *Ada/C to Behavioral VHDL Translator Top Level Design Specification*, JRS Document No. REQ.0179-01, February 10, 1995. Defines the top level design for each of the Translator software modules, including the inputs, processing, and outputs, and a detailed specification of the interfaces.

6. *Ada/C to Behavioral VHDL Translator User's Manual*, JRS Document No. REQ.0187-00, September 1, 1995. Describes the installation and use of the Translators, their restrictions and limitations in detail, how particular language constructs are translated, and error/warning messages.

7. *Technical Report: Language Issues in the Translation of Ada and C to VHDL*, JRS Document No. TR.0190-00, September 5, 1995. Describes the problems encountered and the insights gained from this project regarding specific language issues, from a VHDL standpoint.

8. *Ada/C to Behavioral VHDL Translator Software Demonstration Report*, JRS Document No. REQ.0191-00, October 15, 1995. Summarizes the results of performing the formal software demonstrations defined in the Demonstration Plan.

In addition, four slide presentations regarding the project were created and presented:

1. *Project Kickoff Meeting Presentation*, made October 13, 1994 at JRS, described the goals, rationale, and overall technical approach for the project.
2. *RASSP Principal Investigator Meeting Presentation*, made January 13, 1995 at the RASSP PI conference in Atlanta, described the goals, rationale, development schedule, and current status of the project.
3. *Design Review Presentation*, made March 9, 1995 at JRS, described the top level design and interfaces of the major software modules.
4. *RASSP Conference Summary and Status Presentation*, slide presentation prepared and submitted in July 1995 for the RASSP Conference.

1.7 Organization

Chapter 2 describes the tasks performed for the project.

Chapter 3 contains a general overview and description of the Translator products developed.

Chapter 4 provides more detailed information on the Ada-to-VHDL Translator.

Chapter 5 provides more detailed information on the C-to-VHDL Translator.

Chapter 6 contains a summary and recommendations for future work in this area.

Appendix A contains a sample Ada primitive and its translation into VHDL.

Appendix B contains a sample C primitive and its translation into VHDL.

Chapter 2

Task Summary

This was a software development project, rather than a study or experiment, and the major tasks performed for this project reflect this orientation. The major tasks, or development phases, were:

1. Planning and specifications.
2. Top level design.
3. Detailed design and implementation of the Ada-to-HIL software module.
4. Detailed design and implementation of the C-to-HIL software module.
5. Detailed design and implementation of the HIL-to-VHDL software module.
6. Unit and integration testing of the Translators.
7. Integration of the Translators with NetSyn.
8. Documentation.

2.1 Planning and Specifications

During the planning and specification phase of the project, different technical approaches to the problem were explored, including hand translations

done by several different methods, and the overall technical approach was decided. A set of four documents was then written, as outlined in the CRDL:

1. Requirements Specification
2. Software Test Plan
3. Software Development Plan
4. Demonstration Plan

Together, these documents defined and described the Translator requirements, the technical approach to be used, restrictions and limitations, testing and validation procedures, a detailed list of tasks and milestones, development schedule, and a list of formal product demonstrations.

The technical approach used in developing the Translators is as follows:

An intermediate form called the HIL (High-level Intermediate Language) was defined as an intermediate step in the translation process. The HIL form is at a relatively high level of abstraction, so that high level language constructs such as declarations, expressions, statements, etc. can be directly expressed. Compilers were developed using the Unix tools Lex (scanner) and Yacc (parser generator) to compile both Ada and C to the HIL form. A single HIL-to-VHDL Translator was then developed, which accepts the HIL form from both frontends and generates behavioral VHDL source code. A runtime library was hand-coded in VHDL to provide some necessary runtime support for the translated programs.

The HIL form, and the frontend compilers, provide for translating the complete Ada 95 and ANSI C languages, including those features, such as tasking, which have no subsequent translation into VHDL. They can be viewed as simply providing an alternate form of the source program which can be analyzed and manipulated more easily than the actual source code. The actual translation into VHDL is performed by the backend Translator module. This module utilizes a data file which defines the translation rules for each HIL statement. Provision was also made for the user to extend this data file to specify certain types of special application-specific translation rules.

In summary, there are three software components which comprise the Translators:

1. Ada-to-HIL Compiler – The component which compiles Ada source code into a high-level intermediate form (HIL).

2. C-to-HIL Compiler – The component which compiles C source code into the same high-level intermediate form (HIL).

3. HIL-to-VHDL Translator – The component which translates the HIL intermediate form to VHDL source code. This module is largely data driven from an associated rules file.

In addition, a Runtime Library component, which provides runtime support for the translated VHDL subprograms, was developed.

The actual Translators are implemented as command scripts which invoke the necessary modules in sequence. The HIL form is normally not of concern to or visible to the user; the intermediate file is automatically deleted by default, so the user sees only the final VHDL source file generated.

2.2 Top Level Design

During the top level design phase, the overall design approach was fleshed out and the key interface, the High-Level Intermediate Language (HIL) was defined in detail. The HIL definition was developed by working from syntax summaries of Ada 95 and ANSI C to define a set of high level statements for Ada and C which would enable the complete syntactical structure of each language to be captured in a form which could be easily manipulated and processed. The separate Ada and C HIL statements were then merged to form a common set which became the actual HIL specification. Example programs in both Ada and C were hand-translated into the HIL form to ensure that any valid Ada 95 or ANSI C program could be expressed in HIL.

Several other tasks were also performed during this phase:

- The VHDL-87 and VHDL-93 language specifications were examined carefully and compared with the Ada 95 and ANSI C language specifications to define a complete list of language restrictions for the Translators. Detailed lists were made of features in Ada and C which had no counterpart in VHDL, as well as semantic differences in features which were supported.

- The most significant problem issues were defined and explored, and design approaches were developed to resolve these issues. For both Ada and C, the most significant problem areas were the overall structure of the generated VHDL code and the handling of global variables. An additional major problem area in C was how to handle pointers and pointer operations.
- The standard IEEE VHDL math library was obtained and tested, and plans were made to integrate this package as the core of the HVT Runtime Library.

This phase was completed with the release of a Top Level Design Specification which described the overall design of each module, addressed the key design issues in some detail, defined the complete list of restrictions, limitations, and semantic differences, and contained the complete definition of the HIL, the key interface between the program components.

Up to this point, the project had been a single person effort. However, once the actual software modules and the interfaces were well defined, it became possible to have three people working on the project concurrently and independently, one on each of the major modules (Ada-to-HIL, C-to-HIL, and HIL-to-VHDL). The actual software detailed design and implementation of these three modules proceeded in parallel from this point.

2.3 Ada-to-HIL and C-to-HIL Compilers

The Ada-to-HIL and C-to-HIL modules are discussed together, since they are identical in design and implementation approach. These modules are compilers developed using the standard Unix `lex` and `yacc` tools which translate Ada 95 and ANSI C, respectively, into the HIL intermediate form. As a starting point for the development, public-domain grammars for Ada 95 and ANSI C were located and downloaded from the Internet. The basic design of each module was to use `lex` and `yacc` to create parsers for these grammars, with the code for most of the reduction rules emitting the appropriate HIL statement.

The key design goals for these "frontend" modules are:

1. These modules have *no* relationship to VHDL. Their function is to parse source programs and translate them into an equivalent form

which is amenable to subsequent analysis, manipulation, and processing.

2. The intent is that these modules be able to process *any* valid Ada 95 or ANSI C source program and produce correct HIL. The language feature restrictions apply only to the subsequent translation of HIL to VHDL. There are *no* language restrictions on the translation of source code to HIL.

3. In the event of syntax errors in the input program, the Compilers should issue the best error messages possible and attempt to continue processing. However, it is recognized that good error recovery is difficult, and the Compilers should not devote a significant effort to this task, since for our purposes we do not expect syntax errors in the input programs.

4. Although it is specifically not a requirement that the Compilers detect semantic errors in the input program, it is nevertheless desirable that they do so to the extent feasible. The user should be informed through an error message of any errors of this type detected by the Compilers, but processing should continue and the HIL output file should be written as if no errors were encountered.

5. These modules are not compilers in the sense that they produce executable object code. They produce rather a form of the source program which other application programs (such as the HIL-to-VHDL Translator) should find easy to process.

6. The input files to the Compilers can contain multiple compilation units or functions. The intent is that these Compilers be able to process any source file which can be processed by a conventional compiler for these languages.

7. The actual input to the C-to-HIL Compiler will be the output of the standard C preprocessor rather than the original source file; therefore, the C-to-HIL Compiler is not required to handle the C preprocessor commands. For the C-to-VHDL Translator as a whole, header files are to be preprocessed and automatically converted into VHDL context clauses.

2.4 HIL-to-VHDL Translator

The HIL-to-VHDL Translator program reads the HIL file generated by either the Ada or C to HIL Compiler, and translates the file into functionally equivalent VHDL source code. Essentially, this module *is* the Translator as seen by the user; the Ada-to-HIL and C-to-HIL Compilers can be viewed as simply preprocessing the source file so that it is easier to manipulate, but not performing any of the actual translation into VHDL.

In particular, this module is responsible for fulfilling all of the Translator requirements as defined in the Requirements Specification: the generated VHDL code must be as readable as possible, detailed error or warning messages must be given for unsupported features, the generated VHDL code must be portable to any VHDL implementation, and so forth. Also, this module must be able to generate an entity/architecture test driver for a translated subprogram.

While the detailed design of the Ada and C to HIL Compilers was constrained to utilize Lex and Yacc, so that the implementation was embedded in the parser rules, the detailed design of the Translator was initially unconstrained. The technical approach decided on for the Translator was to utilize an auxiliary data file which contained a set of translation rules, including one for each HIL statement. The rules specified a template for each statement defining how it was to be translated into VHDL, and also contained checking rules to issue warning and/or error messages if required. With this data-driven approach, the job of the Translator program itself was then to interpret and apply the translation rules, which made the program much more general and flexible than embedding the translation rules directly in procedural code. Another advantage of this approach was that the user could be permitted to add his own application-specific translation rules to handle special case situations by effectively extending the rules file, with no changes to the program.

The design goals of the Translator are:

1. The output VHDL program be readable and easily related to the input Ada or C program. This implies using the same identifier names, data structures, etc. to the extent possible.

2. The Translators must issue clear and detailed error and/or warning messages for unsupported features or assumptions made by the trans-

lator. However, it is not required that the Translators process incorrect Ada/C programs and issue error messages as a compiler would.

3. The output VHDL program should be as portable as possible to a wide range of VHDL systems. The use of VHDL-93 features is controlled by a command line switch setting, and it must be possible to generate output VHDL code which uses only VHDL-87 features.

4. The Translators should have minimal ties to a specific host configuration (hardware platform and compiler), and must have no ties to a specific VHDL implementation, i.e., no VHDL implementation dependent features may be used in the generated code.

5. The translator must have the capability of automatically generating a VHDL entity/architecture wrapper, or test driver, for translated programs as well as the programs themselves. This test driver can be used both by the user and during the Translator development to validate the translated program.

2.5 Testing

The goal of the testing phase is to execute a set of tests and test procedures which are necessary and sufficient to provably demonstrate that the Translators are functioning correctly. Correct operation of the Translators is defined as follows:

1. The Translators must detect the use of unsupported features in Ada and C source programs, and output appropriate warning and/or error messages. The output VHDL file is still created in the presence of unsupported features, for manual correction by the user. This is done by including the untranslated statements intact as comments in the output program.

2. For input programs which use only supported features, the Translators must translate the Ada or C input program into a functionally equivalent VHDL source program which compiles correctly using a VHDL compiler. If the Translator option to generate VHDL-87 code is set, then the output VHDL program must compile correctly using any VHDL compiler which supports either the full standard VHDL-87 or

VHDL-93 language; otherwise, it need only compile with a VHDL-93 compiler implementation. "Functionally equivalent" means that the VHDL program computes identical results to the original Ada or C program given the same inputs, within a small tolerance for floating point results.

A secondary goal of the testing process is to evaluate the output VHDL code with respect to its clarity, readability, and its ability to be related back to the original source program, and to improve these characteristics to the extent possible. Since no code optimizations are being performed during the translation process, the performance of the output code will not be evaluated.

The Test Plan document describes the test programs and procedures in detail. To summarize, functional testing is performed on three levels:

1. Unit Testing. Unit testing tests each of the software modules as a separate unit. In this case, individual unit testing was performed on each of the modules: Ada-to-HIL, C-to-HIL, and HIL-to-VHDL.

2. Integration Testing. Integration testing tests combinations of software modules which operate together. A complete Translator consists of either the Ada or C to HIL Compiler, the HIL-to-VHDL Translator, and the Runtime Library. These combinations of software modules are tested and validated.

3. System Testing. System testing involves incorporating a collection of units within a larger system and testing the entire operation. In this case, system testing is performed for the Translators as integrated into NetSyn and the RASSP AST.

There are two main classes of functional tests:

1. Functional correctness tests. These test programs verify that the Translators generate correct code for a wide variety of test programs.

2. Error tests. These tests verify that the Translators correctly detect unsupported features in input programs and issue appropriate error or warning messages.

Functional correctness testing is applicable at all three levels, but error testing is most applicable to the integration test level. At the unit test level only the HIL-to-VHDL Translator should detect unsupported features, since the Ada and C to HIL compilers should correctly translate any valid input program to the intermediate form.

The primary functional test suite for the Ada-to-VHDL Translator is the set of Ada functional primitives from the NetSyn Reusable Software Subsystem (RSS) library, and are actual primitives used in application graphs. This set of primitives was originally obtained from the Navy and is termed the Q003 primitive set. The correct translation of the primitives in this library to VHDL enables complete VHDL behavioral simulations of synthesized networks to be performed and also serves as the Acceptance Test for the Ada-to-VHDL Translator.

For the C-to-VHDL Translator, the primary functional tests are C functions which constitute the Mercury Computer Systems' Scientific Algorithm Library (SAL). These functions have also been added as primitives to the RSS library and may be used in application graphs. As with the Ada primitives, the correct translation of the primitives in this library to VHDL both enhances NetSyn and also serves as the Acceptance Test for the C-to-VHDL Translator.

To broaden the scope of functional testing, and to provide the error test cases, the Ada-to-VHDL Translator also utilized the standard ACVC test suite. Although not many tests from this suite translate correctly (a quick examination shows most test programs use unsupported features such as exceptions, generics, etc.), those that do pass will test the ability of the Translator to handle language features which are not tested in the Q003 primitives, such as string handling; while those that do not will test the error generation capability.

Although there is no direct counterpart to the ACVC test suite for C, there are publically available C test and/or benchmark programs, such as the Stanford Benchmark Suite, which will be used.

The test driver programs generated automatically by the Translators serve as the "main program" for the VHDL simulation, and the simulation serves as the execution of a test. The test drivers contain result checking code, so that each test program generates a pass/fail indication. Most testing is done using automated command scripts to execute an entire test suite sequentially, in a batch mode. The output is directed to a log file, and a

summary test report is generated.

2.6 Integration with NetSyn

Once the Translators were completed as standalone products, the next step was to integrate them into NetSyn. This involved three main tasks:

1. Integration of the Translators with the Reusable Software Subsystem (RSS).
2. Support for the Translators in the NetSyn GUI.
3. The design and development of the functional simulation procedures in NetSyn.

The integration of the Translators with RSS was complicated by the fact that the C language primitives (SAL library) were incorporated into RSS by using an Ada package specification with the body written in C. A

```
pragma interface (C, prim-name);
```

statement was used to define the actual primitive, and it was called from Ada using the `'Address` attribute to pass the addresses of the parameters.

However, the Translators did not support either the interface pragma or the `'Address` attribute, neither of which is available in VHDL, so all of the Ada wrappers for the SAL primitives failed to translate. To correct this problem, the Ada-to-VHDL Translator had to be modified to recognize this situation and generate acceptable VHDL code which would permit the translated Ada wrapper to call the translated C primitive. Because of the strong typing rules in VHDL and the different data types used in the Ada and C programs, this posed a serious problem; it was eventually resolved using a hand-written runtime support package.

The largest part of the integration task was the design and development of the functional simulation procedures in NetSyn. A document describing the requirements and design issues involved was written first. The primary design issues are:

1. How to interface the application graphs to the underlying primitives. This involves such issues as naming conventions and parameter ordering.

2. How to perform multiple functional simulations on different data. This involves issues such as how to support expressions computed at run-time as parameters and how to determine output queue amounts.

3. How to maintain the integrity of the VHDL library. This involves such issues as multiple users of a VHDL library and how to ensure that the version of a primitive compiled in the VHDL library corresponds to the one being used in the graph.

2.7 Documentation

The majority of the project documentation was created near the beginning of the project (*Requirements Specification, Test Plan, Software Development Plan, Software Demonstration Plan, Top Level Design Specification*) and near the end of the project (*User's Manual, Technical Report on Language Issues, Demonstration Report, Final Report*). Throughout the project, monthly status reports and funds expenditure reports were submitted. Also, several slide presentations were created, as well as internal memos regarding problem issues and solutions. A complete list and description of the documents created for the project is contained in Section 1.6.

Chapter 3

Translator Product Descriptions

This chapter describes the operation and environment of the Translators. Both the Ada and C to VHDL Translators are described together, since they are virtually identical. Minor differences are noted in the discussion. The following chapters then discuss the language-specific characteristics and limitations of each Translator separately.

3.1 Operation Summary

The input to the Translators is a source file coded in either Ada 95 or ANSI C. The full languages are supported for parsing, although some program constructs or statements may not be translatable into VHDL. Ada input files may contain any number of compilation units. For C input files, the user has a choice of preprocessing header files or not.

The output from the Translators is a behavioral VHDL program which is logically equivalent to the input file, in the sense that a VHDL compilation and simulation of the translated file will produce the same results as an Ada or C compilation and execution of the original source file. Much of the program structure, data structures, identifier names, and even comments from the input program are preserved, so it is relatively simple to relate the output program back to the input.

However, the output program is structured so that any input file which is translated results in a VHDL output file which can be directly compiled in VHDL. Since VHDL does not permit subprograms as compilation units, procedures and functions which are translated are encapsulated in a package. The package has the same name as the subprogram, so that Ada context clauses work just as in an Ada environment. The Ada Translator supports a rudimentary library system for context clauses, so that translating a set of Ada programs has the same order-of-translation requirements as compiling them with an Ada compiler, i.e., WITH'ed units must be translated first. For C programs, each input file is translated into a single VHDL package: statements outside functions are placed in the package specification, along with a subprogram declaration for each function. The package body contains the subprogram bodies.

The translated program is behavioral VHDL only (the source programs contain no information from which to generate structural VHDL). Further, only sequential code is produced by the Translators; no signals are defined and no concurrent statements are generated. Thus, in VHDL terms, the output is entirely passive code.

Following translation, the VHDL output file is processed just as any other VHDL source file: it must be compiled and otherwise processed by a VHDL system prior to running a VHDL simulation. The VHDL source file produced by the Translators is portable and independent of a specific VHDL implementation. A command line switch specifies whether VHDL-87 or VHDL-93 code is to be generated; however, the differences are currently very minimal. The VHDL-87 generated code should compile correctly using *any* full VHDL 1076 system implementation, including those for VHDL-93, although the reverse is not true.

3.2 Command Lines

The `ada2vhdl` command executes the Ada-to-VHDL Translator, and the `c2vhdl` command executes the C-to-VHDL Translator. The command line formats are almost identical, the only difference is that `ada2vhdl` supports the `-n` option regarding function conversions, and `c2vhdl` supports the `-p` option regarding include files. The command line formats are:

```
% ada2vhdl [-dehiknqsv] [-o vhdl_outfile] [-t subprog-name]
          [-u user_data_file] [-w working_dir] infile
% c2vhdl   [-dehikpqsv] [-o vhdl_outfile] [-t subprog-name]
          [-u user_data_file] [-w working_dir] infile
```

The following command line options set various flags and switches and require no arguments.

`-h` Help switch. Produces command line help information on standard output and exits. If this switch is present, all other command line arguments are ignored.

`-e` VHDL93 switch. Causes VHDL code to be generated that is compatible with the VHDL 1076-1993 Standard. By default, code is generated which is compatible with the VHDL 1076-1987 Standard. Currently, the only difference is that in VHDL-87 the `'IMAGE` and `'VALUE` attributes are not supported; however, future releases may have more significant differences. The code generated for VHDL-87 should work properly with any VHDL implementation, including a VHDL-93 implementation, although the reverse is not true.

`-n` *(Ada only)* Noconvert switch. Overrides the default conversion of Ada functions into VHDL procedures and instead translates them into VHDL functions. See the discussion in Section 4.3.1 for a complete explanation of this switch.

`-p` *(C only)* Causes the Translator to generate inline VHDL code for all of the statements in header files `#include`'d by the preprocessor. By default, VHDL context clauses are generated for header files.

`-s` Include source. Specifies that the lines of the original source program be interspersed in the VHDL output file as comments.

`-q` Quiet mode. In this mode, all translator warning messages are suppressed, both those written to standard output and those included in the VHDL output file. This switch should be used with caution since useful warning messages may be suppressed.

`-v` Verbose switch. Causes the Translator to print summary information including the date and time, input and output file names, and an error summary.

The following command line options also set various flags and switches and require no arguments. However, these options are used primarily for debugging and maintenance purposes; the user will seldom if ever utilize these options:

`-d` Causes the Translator to output various debug messages both to standard error and standard output. Some debug information may also be included in the generated VHDL file, so that it may not be compilable. The format and content of the debugging information is not documented for the user.

`-i` Produces the HIL intermediate file only and then exits; does not produce VHDL code. This switch is commonly used in conjunction with the -d switch for debugging purposes.

`-k` Keep the HIL intermediate file; by default, it is automatically deleted.

The following command line options require a parameter value:

`-o VHDL_outfile` Specifies the name of the VHDL output file. If this option is not specified then it defaults to infile.vhd in the current default directory. If the name is a directory name (the last character is '/'), then the default file name is used (infile.vhd), and the file is placed in the specified directory.

`-t subprog_name` Specifies the name of a subprogram for which a VHDL test driver is to be automatically generated as part of the translation process. `subprog_name` must be the name of a procedure or function in the input source file which is to be tested. Only a single test driver can be generated in each invocation of the Translator. The test driver is generated into file `subprog_name_shell.vhd` in the working directory.

`-u user_data_file` Specifies an optional data file which allows users to customize the translation process with certain types of application-specific translation rules, to handle special case situations.

`-w working_directory` Specifies a directory name of a working directory which contains the intermediate HIL file generated by the Translator. If not specified, it defaults to the user's current default directory.

The `infile` parameter is positional and must be the last item on the command line. It specifies the full file name of the Ada or C source code file which is to be translated. If no file extension is specified, it defaults to `.ada` for the Ada Translator and `.c` for the C Translator.

3.3 Messages

If the Translators encounter any language statements or constructs which cannot be translated into VHDL, or for which the translation is "close" but slightly different from the original program, they will issue error and/or warning messages.

Error messages are issued for statements which cannot be translated to VHDL, in most cases because no equivalent VHDL construct exists, such as for an exception handler in an Ada program. The Translator prints the source line number where the translation error occurred, the source line itself, and the reason for the error (although in most cases, the reason is simply "Operation not supported"). In this case, either no VHDL code is generated for the statement, or the Translator will make a "best guess" translation, but in either case, the source program statement itself is inserted into the VHDL output file at that point as a comment. This permits manual correction to the output file for errors that the user can correct.

Warning messages are issued for statements which can be translated to VHDL, but for which the operation or semantics of the translated program are slightly different from those of the original program. For example, Ada private types are declared as non-private in VHDL, because VHDL does not support private types, or a "double" variable in C is declared as a simple float, since VHDL does not support double precision. The Translator prints the source line number on which the warning occurred, the source line itself, and the reason for the warning. In this case, the Translator does generate VHDL code which in most cases is acceptable as being "the best that can be done". However, it also inserts the source program statement into the VHDL output file at that point as a comment. This permits the user to easily examine the output file and determine whether the warning can safely be ignored. Since this can be tedious in some cases (e.g., numerous warnings about C "long" variables being declared as "integer"), a switch is provided which will disable the generation of warning messages. However, the user should be cautious in using this switch, since some warning messages may

thus be masked which should not be ignored in specific circumstances or applications.

For the C-to-VHDL Translator, the input file is first passed through the standard C preprocessor. Therefore, the line numbers for messages from translating C programs refer to line numbers in the expanded file after including all "#include" files, rather than to line numbers in the original C source program.

3.4 Context Clauses

All VHDL output files contain the context clause:

```
use work.HVT_Runtime_Library.all;
```

The HVT_Runtime_Library is a VHDL package which contains runtime support routines for the Translator; i.e., the translation of some statements involves a runtime call to a routine in this package. Also, all output files from the C-to-VHDL Translator contain the context clause:

```
use work.C_Data_Types.all;
```

This package defines some types used to translate C pointer references.

Normal context clauses in Ada programs (WITH and USE statements) are translated into equivalent context clauses in VHDL. By default, C header files are assumed to be preprocessed, and are translated into context clauses in VHDL. However, the user has the option to expand header files inline; see Section 5.2 for details.

3.5 File Naming Conventions

The files produced by the Translator are assigned names derived from the base name of the input file, e.g., `infile` from `/a/b/c/infile.c`.

The intermediate HIL file is named `infile.hil` and is placed in the working directory, or the default directory if no work directory is specified. This file is automatically deleted after use unless the `-k` switch is specified.

If the `-o` switch is not specified on the command line, the output file containing the VHDL source code is placed in file `infile.vhd` in the default directory. If the `-o` switch is specified and is a file name, then the output file is assigned that name. If the `-o` switch is specified and is a directory name, then file `infile.vhd` is placed in that directory.

In all cases where a file name is specified, it can be a full file specification, but cannot contain environment variables.

If a VHDL test driver is requested (using the `-test subprog-name` option), it is written to file `subprog-name_shell.vhd` in the default directory.

3.6 Environment Variables

The Translator software uses several environment variables, which should normally be defined in the user's login script. They are:

`HVT_BIN` Specifies the directory containing the Translator software; this is the same for all users and is determined when the software is installed.

`HVT_ETC` Specifies the directory containing the Translator support software and data files; this is the same for all users and is determined when the software is installed.

`HVT_LIB` (*Ada only*) Specifies a working directory used by the Ada-to-VHDL Translator as a rudimentary library in which global context files for translated units are placed. This is normally a separate directory created by each user for this purpose. The Ada Translator will write a small text file named `package-name.globals` in this directory for each Ada package specification translated, and read this file when the package is subsequently used in a context clause.

3.7 Runtime Library

Any subprogram calls in the original Ada or C programs are translated "straight across" into VHDL subprogram calls, except that function calls to local functions are normally converted into procedure calls (see Sections 4.3.1 and 5.4.1). Calls to external functions, including standard library functions,

remain as function calls in both Ada and C. Most of these library routines are *not* provided with the Translators, for reasons of project scope, development resource limitations, and the fact that many library routines utilize unsupported features.

Although programs which call standard library routines may be successfully translated into VHDL, the VHDL program will not compile correctly because of the missing library units, and therefore such programs cannot be simulated. The user, of course, always has the option of providing user-written versions of any missing library routines, either directly in VHDL or translated from Ada or C just as any other program. These can be either functional implementations or dummy routines; the latter may be useful to get some translated VHDL programs to compile correctly, if the missing library routines are not critical to the functionality.

The runtime library routines which are included with the Translators fall into three categories:

1. Math Routines: The standard log, exponential, and trig functions are provided, through the VHDL IEEE standard math library package. However, neither the Ada nor C standard math libraries maps directly onto this package; there are differences in names for some of the constants and functions, and both Ada and C provide additional routines not included in the VHDL package. For this reason, the Translators provide two interface packages which serve to interface the math library usage in the RSS primitives with the standard VHDL math package. These packages are:

   `prim_math_lib` This interface package provides a set of math routines to the Ada primitives which utilize the `Common_Data_Types` package.

   `math` This interface package provides C programs with a subset of the standard C `<math.h>` objects, types, and functions. It was created by preprocessing the standard `math.h` header file and then editing it to avoid the use of untranslatable items, such as a pointer to an error function, and to limit the actual math routines to those available in the VHDL math package.

2. Operator Routines: The operator routines provide support for C bitwise operations which are not native functions in VHDL (note that VHDL does provide builtin support for all of these operators for type

BIT_VECTOR, but not for type INTEGER, which is what is needed). There are six routines in this category:

LIB_AND Implements the C '&' (AND) operator.

LIB_OR Implements the C '|' (OR) operator.

LIB_XOR Implements the C '^' (XOR) operator.

LIB_NOT Implements the C '~' (NOT) operator.

LIB_SHL Implements the C '<<' (left shift) operator when VHDL-87 compatibility is requested (VHDL-93 supports shift operations directly).

LIB_SHR Implements the C '>>' (right shift) operator when VHDL-87 compatibility is requested (VHDL-93 supports shift operations directly).

Calls to these routines are generated automatically by the Translator when the corresponding C operation is encountered.

3. Input/Output Routines: The Translator currently supports only two output routines: put_line and new_line; a more complete text input/output facility may be supported in a future release. Users who require more I/O support may provide their own VHDL versions of Ada and/or C standard text input/output routines which can be implemented using the standard VHDL TEXTIO package. The simplest approach is to provide a set of routines which have the same name and calling sequence as the Ada and C I/O routines, and are implemented in VHDL through calls to routines available in the VHDL TEXTIO package.

3.8 User Customization Facility

The Translators provide a facility for users to add certain types of application-specific translation rules, by creating a rules file in a specific format and specifying the file name to the Translator on the command line using the -u switch. This facility is intended to simplify and resolve some problem issues which can arise in translating libraries of Ada or C routines – typically, a large number of routines which utilize a group of common data types, include files, etc.

The rules that can be specified in a user customization file are:

`translate` Translate one string to another wherever it occurs as a token in the input file. The translation is case-insensitive for Ada and case-sensitive for C.

`access_type` (*Ada only*) Specifies the name of a type which is an access type or a structured type such as a record, which contains an access type. In VHDL, subprogram parameters of an access type, or containing an access type, cannot be of mode `in`. As the Translator processes subprogram declarations, any parameters of a type specified in an `access_type` rule which are of mode `in` are automatically changed to mode `inout`, which will enable the output file to compile correctly.

`unconstrained_array_type` (*Ada only*) Specifies the name of a type which is an unconstrained array type. This is used as follows: By default, the Translator converts local functions into procedures (see Section 4.3.1). In order to do this, it must declare a local variable of the return type of the function to hold the result. However, if the return type of the function is an unconstrained array type, it cannot be converted to a procedure since the variable used to hold the result must be constrained when it is declared, but the proper bounds are not known. In general, using a "large" array will not work in these cases, since the result is often processed using attributes of the return value, such as `'Range`, which would then be incorrect. Therefore, when functions are being converted to procedures, if the return type of the function is specified in an `unconstrained_array_type` rule, the Translator will leave that function as a function rather than convert it to a procedure. If such a function violates any of the restrictions for a VHDL function, then it cannot be translated successfully.

`header_file` (*C only*) The format of this rule is: `header_file("name")`. This rule causes the Translator to insert a context clause of the form: `use work.name.all;` in each file translated, as if the C file had done an include of this file. Note that the file is not actually used as a C header file, only the context statement is generated. This rule is intended to be used in special situations where the user has perhaps hand-written some support routines in a VHDL package, and a number of C files are to be translated that utilize these routines.

The name translation rules are applied first, so that if, for example, a name is translated and is also an access type, the translated name must be specified in the `access_type` rule rather than the original name.

3.9 Test Driver Generation

In addition to generating VHDL source code, the Translators also provide the capability to generate a test driver program for the translated code. The `-t subprog-name` command line switch specifies the name of a subprogram for which a VHDL test driver is to be automatically generated as part of the translation process. A test driver can only be generated for a procedure, so `subprog-name` must be the name of a procedure (or a function which was converted to a procedure) in the input source file. If the source file contains a package, then the subprogram must be one of those defined in the package. Only a single test driver program can be generated in each invocation of the Translator.

The overall structure of the test driver program is as follows: It consists of a single VHDL source file containing one entity and one architecture for that entity. The name of the entity is `subprog-name_Entity` and the name of the architecture is `subprog-name_Test`. The entity has a set of generic parameters corresponding to the parameters of the subprogram, so that multiple tests can be run using the same test shell by setting the actual test values on the command line when a simulation is executed. The architecture contains a process, in which local variables are defined corresponding to each subprogram parameter, and are initialized using the value of the corresponding generic parameter. The process initializes the input parameters, calls the subprogram, and compares the computed results to the expected results. The test driver makes calls to routines in a support package named `HVT_Test_Support` to read input datasets, print out values, and make comparisons. The test driver prints the results to standard output. Note that the test driver, like the translated code, does not contain any signal definitions; hence no report file is created, and the simulation executes in 0 simulated time.

Due to the strong typing rules in VHDL, the runtime support package which provides the support routines to read datasets, print values, and compare results must know the datatypes involved in the subprograms being tested. Further, the generation of the test driver also requires some "builtin" knowledge of the data types: which are scalars or arrays, how many dimensions, etc. The bottom line is that test drivers cannot be automatically generated for arbitrary data types. The Translators currently support only standard integer and real data types, plus the special data types used in the RSS Q003 and SAL primitive libraries. In the current release of the Translators,

the test driver generation mechanism cannot be extended by the user to support application-specific data types.

Chapter 4

Ada-to-VHDL Translator

4.1 Overview

The input to the Ada-to-VHDL Translator is a single Ada source file, the same as would be input to an Ada compiler. The file may contain any number of Ada compilation units. The output is a single VHDL source file, which in general will contain multiple VHDL compilation units. The output program is structured so that any input file which is translated results in a VHDL output file which can be directly compiled in VHDL. Since VHDL does not permit subprograms as compilation units, procedures and functions which are translated are encapsulated in a package. The package has the same name as the subprogram (which VHDL does permit), so that Ada context clauses work just as in an Ada environment. The Ada Translator supports a rudimentary library system for context clauses, so that translating a set of Ada programs has the same order-of-translation requirements as compiling them with an Ada compiler, i.e., `WITH`'ed units must be translated first.

4.2 Restrictions and Limitations

The following Ada 95 language features are not supported by the Translator, and will cause an error message to be generated if they are used in the input program. These restrictions are due to the fact that there are no

straightforward implementations of these features in the VHDL language. Many of these features are those added to Ada in the Ada 95 language revision.

- Tagged types and type extensions.
- Stream types including stream input/output.
- Abstract types and subprograms.
- Protected types, units, objects, and actions.
- Fixed point types.
- Modular (unsigned) integer types.
- Discriminated types.
- Variant records.
- Initial values for record fields declared in the record type definition.
- Strings without an explicit length that are initialized from expressions.
- Access-to-subprogram definitions.
- Aliased objects.
- Representation clauses.
- Machine code insertions.
- Exceptions and exception handlers.
- Generics.
- `GOTO` statements.
- Renaming declarations.
- Nested package declarations.
- Initialization code in package specifications.
- Subprogram or package declarations within a block.
- Subunits.

- Tasking.
- All attributes *except* the following: `'Base`, `'First`, `'Image`, `'Last`, `'Length`, `'Pos`, `'Pred`, `'Range`, `'Succ`, `'Val`, `'Value`.
- User-defined assignment and finalization.
- Use-type clauses.
- Hierarchical library units (e.g., child packages).
- Unchecked type conversions and unchecked deallocation.
- Wide characters including wide text input/output.
- All annexes.

The following Ada 95 language features are not supported exactly by the Translator, in the sense that the semantics are identical to those of the Ada program, but are translated into approximately equivalent VHDL code as described, which will be acceptable for many applications. A warning message is issued by the Translator when these statements are encountered.

- All pragmas are ignored.
- Private types, including limited private, are treated as normal types.
- Short integer and floating point types are translated into standard single precision. Long integer and floating point types are also translated into single precision, but a warning message is issued.

Also, none of the standard packages or child packages in the predefined language environment, including `TEXT_IO`, `CALENDAR`, `SYSTEM`, etc. is provided as a builtin part of the Translator. References to these packages are not errors, per se, since they will result in the generation of VHDL context clauses of the same name. However, the user is responsible for creating the VHDL units necessary to compile and/or simulate programs which utilize these packages.

4.3 Translation Notes

This section describes how certain Ada statements are translated, and discusses some issues and consequences of the approach taken.

4.3.1 Functions vs. Procedures

Since Ada and VHDL both support function and procedure subprograms, it seems that the natural choice is to just translate the Ada subprogram structure directly into VHDL. However, function subprograms in VHDL have two significant restrictions:

1. A function parameter cannot be of an access type, or a composite type, such as a record, that contains an access type.
2. Functions cannot access global or uplevel variables.

A significant number of Ada functions cannot be translated "straight across" into VHDL functions without violating these restrictions. Therefore, the Translator supports two ways of processing function subprograms: they can be left as functions or converted into procedures (which do not have either of these restrictions). Obviously, if a function subprogram is converted to a procedure, then calls to that function must also be converted into procedure calls.

By default, the Translator converts most functions into procedures, to try to ensure that the translated code is valid VHDL. The conversion is done by appending an additional parameter to the end of the parameter list to hold the return value. The name of this parameter is `function-name_Result`, its type is the return type of the function, and its mode is `out`. Return statements within the function are modified to assign the return value to this parameter and then return. Calls to these functions are modified as follows:

1. A temporary variable of the correct type is declared to hold the result of each function call. The name of the temporary is `function-name_n`, where `n` is a sequential number starting at 1.

2. The function call is changed into a procedure call statement, with the temporary variable added as the last parameter, and the other parameters exactly as they were in the function call.

3. The procedure call is made immediately prior to the statement in which the function call is used; if multiple function calls occur within a single statement, the corresponding procedure calls occur in the normal order of expression evaluation.

4. The temporary variable containing the function result is then used in place of the function call in the expression in which it appeared.

The resulting VHDL code is somewhat less sleek than the original Ada code, but is logically equivalent, and, if the function violates either of the VHDL restrictions, is necessary in order to produce valid VHDL code.

There are some additional issues related to the function conversions:

1. If a function call occurs in the initialization of a declared variable, the corresponding procedure call cannot be made at that point. Therefore, the variable is declared only, and the initialization is postponed until the beginning of the executable code segment.

2. If a function returns an unconstrained array type, it cannot be converted to a procedure since the temporary variable used to hold the result must be constrained when it is declared, but the proper bounds are not known. In general, using a "large" array will not work in these cases, since the result is often processed using attributes of the return value, such as `'Range`, which would then be incorrect. Therefore, even when functions are being converted to procedures, if the Translator knows that the return type of a function is an unconstrained array type, it will leave that function as a function rather than convert it to a procedure. This information can be provided through the user customization file; for more information see Section 3.8.

3. In the general case, it can be quite difficult to determine the return type of a function declared in another compilation unit and referenced through a context clause. Therefore, only functions (including nested functions) which are declared within the current compilation unit are converted to procedures. This also avoids the common situation where functions from a standard library, such as the math library, are used.

These must remain as function calls in order for the program to work properly.

4. Overloaded functions may not work properly when converted to procedures, although overloaded procedures and overloaded functions which are not converted to procedures are supported.

The user may choose to override the default conversion of functions to procedures by specifying the `-n` switch to the Translator. If this switch is set, then all Ada functions and function calls are translated into VHDL functions and function calls "straight across"; variable initializations are performed as specified, etc. This method will generally produce simpler and more readable VHDL code if the converted functions do not violate any VHDL restrictions.

4.3.2 Global Variables

Global variables in Ada are those declared in package specifications. These variables can be accessed by any compilation unit that `WITH`'s the package, or by any subprogram in the package body. It is relatively common in Ada, at least for larger programs, to use global variables for sharing data and for communication between program modules.

VHDL-87 does not support object declarations in package specifications (or bodies). In VHDL-93, the concept of *shared variables* which could be declared in package specifications was added, but even these do not have the generality, and are not intended to be used for the same purpose as Ada global variables. In fact, there is no equivalent mechanism in VHDL subprograms or packages to utilize global variables. The `process` statement provides such a mechanism, but the use of concurrent statements is outside the scope of the Translator.

In order to translate programs which utilize global variables, the Translator uses the following approach:

1. Whenever a package specification is translated, the Translator creates a small data file in a library directory which defines the global variables and the context of that package. This is a simple form of the traditional library mechanism used by Ada compilers. The file is named `package-name.globals`. Note that this is only done for package specifications, and that a `.globals` file is always created for a package,

even if it is empty. Any objects defined in the package specification (which are global variables in Ada) are commented out in the translated package specification, since they would not compile in VHDL. The comment has a recognizable form which begins with the string `--*GLOBAL*--`.

2. Whenever the package is referenced in a context clause, the Translator reads the `.globals` data file, and inserts any object declarations made in the package into each subprogram within the scope of the context clause as *local* variables. The variables are flagged with a recognizable comment of the form `--*GLOBAL*-- from package-name`. This permits the translated programs to compile correctly, since the objects within all of the context clauses are defined, although as local variables and not global.

3. Since variable declarations in package bodies are also prohibited in VHDL, any such declarations are also commented out and inserted inside each subprogram in the package body similar to other global variables. However, since in Ada, the scope of these variables is limited to the package body, there is no need to create a `.globals` file for them.

4. If a `.globals` file is not found for a unit referenced in a context clause, a warning message is issued. This can arise in two situations:

   (a) If the user has violated the order-of-translation rules by translating a unit prior to a package on which it depends, then the warning message is valid, and the user should translate the package and then re-translate the unit which gave the warning message.

   (b) If the unit referenced in a context clause is a subprogram instead of a package, then no `.globals` file will exist for it (recall they are only created for packages) and the warning message can be ignored. However, this case will still work properly, because the subprogram will have been translated into a VHDL package of the same name, so that the context clause is still valid in VHDL.

It is very important to understand that the VHDL code produced by the Translator to handle global variables only permits each individual translated module to compile correctly. A complete program which utilizes global variables will not work properly unless the modules are "linked" to move the global variable declarations out of each subprogram and merge them in a

single VHDL global context such as a `process` *statement, prior to simulation. The creation of a complete "linked" VHDL program in this manner is beyond the scope of the Translator, but the* `--*GLOBAL*--` *flags are intended to aid the user in this task.*

4.3.3 Blocks

Although Ada and VHDL both support program structures called "blocks", they are much different concepts. In Ada, a block defines a declarative region of the program for scope and visibility purposes, and may have local objects declared. In VHDL, a `block` statement is a concurrent statement which groups together other concurrent statements which represent a portion of a design and is intended to support design decomposition. In particular, in VHDL blocks may not occur within a subprogram.

The Translator supports blocks in Ada programs by moving any block-local type definitions and/or variable declarations to the subprogram level, with the variable names changed to avoid potential conflicts. Subprogram or package declarations within a block are not supported. After the declarative part of the block is moved, the block boundaries are effectively ignored. The names of block-local variables are changed to `name_n` where `n` is a sequential number starting at 1, and are flagged in their declarations with the comment `-- Block local variable`. In most cases, this procedure results in an acceptable translation, and is the closest we can come to reproducing the Ada block concept in VHDL.

However, it is very important to recognize that there is a potential problem in the translated VHDL code which the Translator cannot detect, which can arise in the case where the definition or initial value of a block local variable depends on computations performed in an enclosing region prior to the elaboration of the block. In this situation, moving the declaration of the variable to the start of the program does not have the same effect, because the value(s) on which it depends have not yet been computed. However, in this case, there is in general no way to translate the program correctly to VHDL, since VHDL has no mechanism for elaborating declarations during the execution of a subprogram.

As an example, consider the Ada program fragment:

```
NT : Integer := 1;
...
NT := 2*NR + 1;
...
declare
  C : Vector (1..NT);
begin
  for i in C'Range loop
  ...
```

The VHDL translation for this program will result in code that looks like:

```
NT : Integer := 1;
C_1 : Vector (1..NT);
...
NT := 2*NR + 1;
...
for i in C_1'Range loop
  ...
```

Although this code will compile correctly, it will not simulate correctly because the bounds of array C are not correct when the declaration is moved ahead of the computation of NT. However, there is no feasible way for the Translator to detect this type of problem, or to generate correct VHDL code for this instance.

4.3.4 Constants

It is common practice in Ada to name literal values using `constant` declarations, and then use the name in place of the value. VHDL also supports `constant` declarations for literal values, but with a significant difference: Ada constant declarations define literals of a universal type, whereas in VHDL, an explicit type must be specified in the constant declaration.

When an Ada `constant` declaration is encountered, the Translator handles it as follows:

1. If the constant value is a literal and does not contain a decimal point, then it is translated to VHDL as a constant integer.

2. If the constant value is a literal and does contain a decimal point, then it is translated to VHDL as a constant real.

3. If the constant value is a not a literal, then its type cannot be determined and the Translator issues an error message.

However, even in the first two cases, a warning message is issued, which says "Type assigned for constant may not match usage". The reason for this is that once the constant has been declared in VHDL as an integer or real, the strong typing rules are enforced on that type. As an example, if an Ada program defines `PI` as a constant, it is valid to assign `PI` to a floating point variable of type `MY_FLOAT`. However, in the translated VHDL program, `PI` will be declared as a constant of type `real`, and the assignment to a variable of type `MY_FLOAT` will cause a VHDL compilation error.

4.3.5 Strings

In Ada, a string declaration can use an initial value to implicitly determine the length of the string, but in VHDL the length must be explicit. The Translator will determine the correct length if the initial value is a simple string, but will issue an error message if no length is specified and the initial value is an expression. For example:

```
a : string (1..5);                    -- OK
b : string (1..5) := "abcde";         -- OK
c : string := "abcde";                -- OK, Translator will determine length
d : string := "abc" & "de";           -- Translator does not handle this case
e : string := ('a','b','c','d','e');  -- Translator does not handle this case
f : string (1..5) := "abc" & "de";    -- OK, because length is explicit
```

4.3.6 Internally Generated Names

In almost all cases, identifier names in the Ada input program are used unchanged in the translated VHDL program, but in a few instances, the Translator generates names internally by appending a string to an existing identifier. In rare cases, the identifier generated by the Translator may conflict with a user-defined identifier elsewhere in the program. The Translator

does not check for this situation, so if it arises, the simplest solution is to modify one of the names to avoid the problem.

The Ada Translator can append the following strings to an identifier:

_n where n is a number starting at 1. This scheme is used for block-local variables and temporary variables created to hold function results.

_Result for the return parameter of a function which is converted to a procedure.

Chapter 5

C-to-VHDL Translator

5.1 Overview

The input to the C-to-VHDL Translator is a single ANSI C source code file, the same as would be input to a C compiler. The file may contain any number of C functions, include files, etc. The output is a VHDL source file containing a single VHDL package specification; the name of the package is the base file name of the input file. Source statements outside functions are placed in the package specification, along with a subprogram declaration for each function. The package body contains the subprogram bodies. All of the C functions in the file are converted to procedures in VHDL.

5.2 Header Files

For C programs, the user has a choice of whether to pretranslate header files and reference them using VHDL context clauses, or to expand the header files inline in each translated program. The pretranslation approach is recommended and is the default, since it is more appropriate to the VHDL environment, and it provides a means to avoid the often lengthy inclusions of header files for translation in each separate C program. When translating a specific C program, all of its header files must be treated the same way, i.e., either all expanded or all referenced as context clauses.

To pretranslate header files, the user simply translates them normally. For

example, to pretranslate a header file named `defs.h`, the user would enter the command:

```
% c2vhdl defs.h
```

This will create an output file named `defs.vhd` containing a VHDL package named `defs` in which the include file is expanded. This file must of course be compiled into the VHDL library prior to any compilations which reference it in a context clause. Subsequently, the inclusion of this header file in other C files would generate the VHDL statement:

```
use work.defs.all;
```

Thus, a partial order-of-translation requirement is introduced for pretranslated header files, in that they must be translated and compiled prior to *compiling* any translated modules which reference them. However, unlike Ada, there is no order requirement strictly on the translation step.

The `-p` command line switch disables the context clause generation and causes the complete bodies of all include files to be included and translated to VHDL along with the rest of the program. This switch should not normally be used, but is provided for special situations.

In some cases, especially for the standard C include files, the header file may not translate completely due to the presence of untranslatable features. In this case, it may be necessary to hand-modify either the include file or the translated VHDL file, or, in extreme cases, to hand-code an equivalent file in VHDL. The Translator includes a partial version of the standard C `<math.h>` header file produced in this way, since that file contains several untranslatable constructs, such as function pointers, and the translated code also needs to utilize the standard VHDL math library routines instead of the C library routines.

To handle special application-specific situations, the Translator user-customization capability also provides a mechanism to add context clauses to the output programs which were not present in the input.

5.3 Restrictions and Limitations

The following ANSI C language features are not supported by the Translator, and will cause an error message to be generated if they are used in the input program. These restrictions are due to the fact that there are no straightforward implementations of these features in the VHDL language.

- Variable length argument lists.
- Abstract declarators in argument lists.
- Explicit values for enumerated type identifiers.
- `GOTO` statements.
- Wide characters and trigraphs.
- The `sizeof` operator.
- Automatic type conversions.
- `switch` statements in which the `default` alternative does not appear last.
- Nested `switch` statements with fall-throughs at an inner level.
- Array subscript references with the index first and the array name in brackets.
- Subarrays, i.e., array references with fewer subscripts than in the array definition.
- Pointers, pointer operations, and the indirect and address operators are partially supported; see the discussion below for details.

The following C language features are not supported exactly by the Translator, in the sense that the semantics are identical to those of the C program, but are translated into approximately equivalent VHDL code as described, which will be acceptable for many applications. A warning message is issued by the Translator when these statements are encountered.

- Unsigned types are treated as signed.

- Short integer types are translated into standard single precision. Long integer and floating point types are also translated into single precision, but a warning message is issued.

Also, the standard C runtime library routines, including memory allocation, I/O, string handling, etc. are *not* provided as a builtin part of the Translator. References to these routines, through the corresponding standard C header files, are not errors, per se, since they will result in the generation of VHDL context clauses of the same name. However, the user is responsible for creating the VHDL packages necessary to compile and/or simulate programs which utilize these functions. One exception to this is that the Translator does provide a subset of the standard `<math.h>` header file, and the math routines are mapped to the corresponding routines in the standard VHDL IEEE math runtime library.

5.4 Translation Notes

This section describes how certain C statements are translated, and discusses some issues and consequences of the approach taken.

5.4.1 Functions vs. Procedures

Since C and VHDL both support function subprograms, it seems that the natural choice is to just translate a C function directly into a VHDL function. However, function subprograms in VHDL have two significant restrictions:

1. A function parameter cannot be of an access type, or a composite type, such as a record, that contains an access type.
2. Functions cannot access global or uplevel variables.

With these restrictions, the majority of C functions would contain invalid code if they were translated "straight across" into VHDL functions. Therefore, the Translator converts all C function bodies into VHDL procedures instead of functions (procedures do not have either of these restrictions). Obviously, if a function subprogram is converted to a procedure, then calls to that function must also be converted into procedure calls.

The conversion of C functions to VHDL procedures is done similarly to Ada; the mechanism is described in detail in Section 4.3.1. However, there are some differences in the C environment:

1. If the return type of the function is `void`, then no extra parameter is added. However, note that if the return type specification is omitted, which is not uncommon in practice, the default type is `int` and not `void`, so that in this case, an extra parameter of type `int` is added.

2. If a function is declared as `extern`, then calls to the function are left as function calls rather then being converted to procedure calls. The reason for this is that most often these functions are from a standard library, such as the math library, and if they were converted to procedure calls then the entire library would have to be converted as well, which is very awkward.

3. Note that in the Ada Translator, a command line switch is provided to override the conversion of functions into procedures and instead leave them as functions. This switch is *not* provided in the C Translator; hence C functions are *always* converted into procedures.

5.4.2 Pointers

Because pointers are so important in C programs, partial support for pointers and pointer operations is provided by the Translator even though VHDL does not support pointers directly. The basic rule is that only simple local pointers to scalar types are supported. Thus, parameters or local variables declared as `int *` or `float *` are supported, but constructs such as pointers to functions, arrays, or structures, arrays of pointers, functions returning pointers, and pointers to pointers are not supported. Global pointers (`static` or `extern`), pointers to global objects, and dynamic pointers (e.g., returned by `malloc`) are also not supported. Although clearly the approach is restrictive, this level of support is adequate to handle a significant number of situations involving common pointer usage.

Pointer support is provided by creating an *associated array* for each pointer, which is a one dimensional VHDL array of the scalar type referenced by the pointer, and indexed by the pointer itself. The name of the associated array for each pointer is `pointer-name_Array`. The name of the associated array must be known when a pointer is dereferenced, or an error occurs.

The association of an array with a pointer operates as follows:

1. If the pointer is a formal parameter, an associated array is created and is used as the parameter instead of the pointer. It is declared as an unconstrained array of the correct type, while the pointer itself is declared as a local variable of type `pointer`[1] which is initialized to the lower bound of the associated array.

2. If the pointer is a local variable, it is declared simply as a pointer and initialized to `nil`. Its associated array is determined subsequently based on pointer assignments in the program. The first time the pointer is assigned, its associated array is set to the same as that of the pointer that it is assigned from. If it is assigned from something other than a pointer expression, or if the associated array of the pointer it is assigned from is unknown, then an error message is issued. Once established, a subsequent association of a pointer with a different array is also flagged as an error, since when the pointer is dereferenced, its current association cannot in general be determined at compile time.

Pointer arithmetic is performed normally, with the pointers being treated as integers. When a pointer is dereferenced, the Translator generates a VHDL array reference for the associated array using the pointer as the subscript.

5.4.3 Address Operator

In general, the address operator (`'&'`) is not supported, since VHDL does not support the `'Address` attribute, and there is no feasible way to emulate this operation. However, in conjunction with the support for certain kinds of pointers using array references, certain restricted kinds of address operator references are also supported using array references:

1. The address of an array used as an argument is replaced by the array itself, since array arguments are passed by reference in VHDL. This concept is extended in the Translator such that arguments of the form

[1]The `pointer` type is simply a subtype equivalent to an integer, and is used for readability. The pointer types are defined in a support package named `C_Data_Types`, and a context clause for this package is automatically generated for every C program translated.

`&array[n]` are translated into an array slice starting at `n`, which has the same effect in VHDL, e.g., `&array[n]` is translated into VHDL as `array(n to array'High)`.

2. If a pointer is assigned from the address of an array, then that array becomes the associated array for the pointer.

5.4.4 Switch Statements

C `switch` statements are translated into VHDL `case` statements. The translation is straightforward as long as each case alternative ends with a `break` statement. However, C permits one case alternative to fall through to the next, while VHDL does not. To handle this situation, the Translator duplicates statements in case alternatives which fall through to make the resultant VHDL code equivalent to the original, i.e., if case A falls through to B which falls through to C, and case C ends with a `break` statement, then in the VHDL code, the statements in cases B and C are duplicated and appended at the end of case A, and the statements in case C are duplicated at the end of case B.

Since `switch` statements can be nested, this process could get quite complex in the general case. Therefore, the Translator supports nested `switch` statements only if the inner `switch` statements have a `break` statement following each alternative, so that no code duplication in the nested statements is necessary.

There is one further restriction: C permits the `default` alternative to be anywhere within the `switch` statement; the Translator only supports the case where the default alternative appears last.

5.4.5 Declaration Qualifiers

Declaration qualifiers are handled as follows:

- `volatile` declarations are treated as normal declarations.
- `auto` variables correspond to normal local variables in VHDL.
- `register` variables are treated as `auto`.

C variables declared as either `static` or `extern`, or declared outside of a function, are equivalent to Ada global variables, and are handled in a similar manner by the Translator; see Section 4.3.2 for a more complete discussion.

Since VHDL does not support global objects in sequential code, the Translator must make these objects local variables in order for the VHDL code to compile. To translate programs which utilize C `static` or `extern` variables, the Translator places the variable declarations in the VHDL package specification, but then comments them out there with the recognizable comment header `--*GLOBAL*--`, since object declarations in a package specification are invalid in VHDL. All of the global variables are then inserted into the translated code for each C function body as local variables, again flagged with the `--*GLOBAL*--` comment. However, since C does not support the direct context references to other files as Ada does, the library mechanism and the `.globals` data file created for Ada programs are not necessary for C.

It is very important to understand that the VHDL code produced by the Translator to handle `static` *and* `extern` *variables only permits each individual translated C file to compile correctly. A complete C program which utilizes global variables will not work properly unless the translated modules are "linked" to move the global variable declarations out of each subprogram and merge them in a single VHDL global context such as a* `process` *statement, prior to simulation. The creation of a complete "linked" VHDL program in this manner is beyond the scope of the Translator, but the* `--*GLOBAL*--` *flags are intended to aid the user in this task.*

5.4.6 External Arrays

If an array is declared as `extern` and its size is not specified, some bounds must be supplied in the VHDL declaration because an unconstrained array cannot be declared as a variable. The bounds supplied by the Translator in this case are `0 to Max_Extern_Array_Dim` where the upper bound is a constant declared in the `C_Data_Types` support package. Setting the correct bounds for such arrays is another issue the user must address in creating a "linked" VHDL program as described above.

5.4.7 Blocks

Although C and VHDL both support program structures called "blocks", they are much different concepts. In C, blocks are used most often to group together a sequence of statements to form a compound statement as required by C syntax, although it may also have local objects declared. In VHDL, a `block` statement is a concurrent statement which groups together other concurrent statements which represent a portion of a design and is intended to support design decomposition. In particular, in VHDL blocks may not occur within a subprogram.

The Translator supports blocks in C programs by moving any block-local variable declarations to the subprogram level, with the variable names changed to avoid potential conflicts. After the local declarations (if any) are moved, the block boundaries are effectively ignored. The names of block-local variables are changed to `name_n` where `n` is a sequential number starting at 1, and are flagged in their declarations with the comment `-- Block local variable`. In most cases, this procedure results in an acceptable translation, and is the closest we can come to reproducing the C block concept in VHDL.

However, it is very important to recognize that there is a potential problem in the translated VHDL code which the Translator cannot detect, which can arise in the case where the definition or initial value of a block local variable depends on computations performed in an enclosing scope prior to entering the block. In this situation, moving the declaration of the variable to the start of the program does not have the same effect, because the value(s) on which it depends have not yet been computed. However, in this case, there is in general no way to translate the program correctly to VHDL, since VHDL has no mechanism for elaborating declarations during the execution of a subprogram.

5.4.8 Strings

C arrays of characters are converted into VHDL strings. However, since C arrays start at 0 and strings in VHDL must start at 1, if any index manipulations are performed on objects of type `array of char`, it is possible that the computed position will be off by 1.

5.4.9 Unions

C unions are translated into normal record structures in VHDL, so that element storage is not overlapped; there is no way to create memory-overlapped data structures in VHDL. In most cases this translation will be adequate, however, if the program makes any assumptions about the actual layout, e.g., accessing the same component as different data types, then using a normal record will not work. However, in this case, there is in general no way to translate such programs correctly into VHDL.

5.4.10 Identifier Names

In most cases, identifier names in the C input program are used unchanged in the translated VHDL program, but in some instances, the Translator may modify the name or generate a derived name internally by appending a string to an existing identifier. In rare cases, the identifier generated by the Translator may conflict with a user-defined identifier elsewhere in the program. The Translator does not check for this situation, so if it arises, the simplest solution is to modify one of the names to avoid the problem. The C Translator can modify or create names as follows:

1. If an identifier begins or ends with an underscore character, or contains two or more consecutive underscore characters, which makes the identifier illegal in VHDL, then these underscore characters are changed to `'X'`.

2. Since identifiers are case-sensitive in C but not in VHDL, if an identifier differs only in case from one already defined, it is modified by repeating the last character. For example, if `I` is defined, and `i` is encountered, then `i` and all references to it are changed to `ii`.

3. If the identifier is a reserved word in VHDL, such as `out`, it is also modified by repeating the last character, i.e., `out` becomes `outt`.

4. For block-local variables and temporary variables created to hold function results, the string _n where n is a number starting at 1 is appended to the identifier name.

5. The name of the return parameter for a function which is converted to a procedure is `function-name_Result`.

6. If a parameter is written to within a function, a local copy of the parameter is created, in order to emulate C semantic rules in VHDL. To minimize name changes, the name of the formal parameter is changed to `name_Parm`, and the actual parameter name is declared as a local variable and initialized to the value of the actual parameter at entry.

Chapter 6

Conclusions and Recommendations

The Government has a significant interest, and stake, in all of the programming languages involved in this project: Ada and VHDL originated as a direct result of Government efforts to reduce costs and provide standardization in software and hardware development projects. C (and also C++) is becoming increasingly important as the Government embraces COTS systems and encourages the development of dual-use technologies.

Further, hardware and software development are no longer separate camps, but are fast merging as hardware/software codesign methodologies and tools emerge which focus on the overall system design problem. The entire RASSP program and its goals reflects these trends.

Hence, the issue of compatibility and interoperability between these languages is, or ought to be, a topic of significant interest. As the boundary between hardware design and software design continues to blur, it becomes increasingly important that algorithms be transportable, or at least easily convertible, between Ada and C, on the software side, and VHDL, on the hardware side. A VHDL version of an algorithm coded in Ada or C can be used to synthesize custom hardware to implement the algorithm, or as part of a VHDL functional simulation of a processor designed to execute the algorithm.

6.1 Conclusions About Language Issues

VHDL-93 includes a foreign language interface capability, which could permit, e.g., a C function to be invoked directly during a VHDL simulation. However, this capability is optional in a VHDL implementation, and is not standardized even among those implementations which do support it. This feature could help solve the simulation part of the problem in a particular instance, but is hard to generalize because of the lack of standardization. However, no synthesis system will produce a VHDL model for a part which is described using the foreign language interface; VHDL source code is needed. Other complementary design and analysis tools also work directly on VHDL code only. Therefore, there is a definite need for a means to perform language conversions automatically, or nearly so. The Ada and C to VHDL Translators are a significant step in this direction, but are limited, as any such tool would be, by the features and limitations of the VHDL language with regard to sequential code as compared with Ada and C.

Clearly, the languages have different purposes, and it is not being suggested that VHDL become a superset of Ada and/or C to facilitate conversions and interoperability. However, there are two significant language features which stand out in importance which would greatly simplify the translation process for both Ada and C:

1. Variable declarations should be permitted in VHDL packages, both in specifications and bodies, with the same semantics as in Ada, i.e., variables declared in package specifications are globally defined, and those in a body are global to the body. The addition of *shared variables* in VHDL-93 may resolve part of this issue, however, this is not entirely clear.

2. The restrictions on VHDL functions should be removed; specifically, functions should not be prohibited from having access type parameters and should be permitted to access global and/or uplevel variables in the same manner as procedure subprograms.

There are also several features of lesser importance which would simplify the translation process:

1. `CONSTANT` declarations should be treated as being of a universal type,

as in Ada, so that the use of such constants does not trigger type clash errors.

2. Block statements, or the equivalent, containing local variable declarations should be permitted in subprogram bodies. However, the generality provided in Ada is not needed, e.g., one need not be able to declare packages or subprograms within a block.
3. The `'Address` attribute would be very useful.
4. The `GOTO` statement would be very useful.

6.2 Recommendations for Future Work

There are three areas in which it would be worthwhile to extend the Translators:

1. Remove some of the restrictions and limitations, especially for the C Translator.
2. Add additional Runtime Library support.
3. Add the capability to manipulate the generated code as part of the translation process.

6.2.1 Removal of Language Restrictions

The capability of the present Ada-to-VHDL Translator is limited primarily by the language features available in VHDL. Not much more can be done to remove the restrictions on translating Ada programs with the current VHDL language. However, there are three issues which could be addressed:

1. The conversion of Ada functions into procedures, which is more likely to result in a successful translation, is currently performed only for local functions, due to the difficulty of determining with complete generality the return type of a function referenced through a context clause. However, this capability could be significantly improved in the Translator by adding the ability to determine the result type of functions declared in a package specification and used in the body, and by performing overloading resolution.

2. It is possible to add a preprocessor program to the Translator which would expand generic instantiations inline. Although this process is more complex than macro expansion, the principle is similar. With such a preprocessor, the restrictions on translating Ada programs which use generics would disappear; this is one of the most significant current restrictions.

3. Add the capability (probably as a separate program) to link complete VHDL programs by processing and merging the lists of global variables in each linked compilation unit.

On the other hand, the present version of the C-to-VHDL Translator has a number of restrictions which are due to resource limitations of the project. Many of these could be removed by additional effort, which would permit the C Translator to successfully translate a much greater class of C programs. A number of potential Translator enhancements fall into this category; some of the most significant are:

1. Handling additional types of pointer operations. Probably the most important of these are dynamic pointers (e.g., calls to `malloc`), arrays of pointers, and pointers to array and structure types.

2. Support for the address operator (`'&'`).

3. Detecting and correctly translating mixed mode expressions by inserting the appropriate type conversions automatically.

4. Recognizing which pointer parameters represent scalar parameters which are modified, and changing them to scalar `out` parameters instead of single-element arrays. This is an awkwardness of the present system which affects program readability and to some extent functionality.

Also, as with Ada, we need the capability to link complete VHDL programs translated from C by processing `extern` objects and functions correctly.

6.2.2 Runtime Library

Although it is probably not worthwhile to attempt to do a complete runtime library for the Translators, it would be very useful to add some additional routines, particularly I/O. Text I/O should be supported in the

Translators by providing VHDL versions of both Ada and C standard text input/output routines, i.e., a set of routines which have the same name and calling sequence as the Ada and C I/O routines, and are implemented in VHDL through calls to routines available in the VHDL `TEXTIO` package.

6.2.3 Code Optimization

Potentially the most important addition to the Translators is the capability to manipulate the generated code as part of the translation process, instead of performing a "straight" translation as is done currently. There are two reasons for doing this:

1. To perform conventional optimizations on the code, much as a compiler would, e.g., loop optimizations, to reduce VHDL simulation time. This may be moderately useful, however, we may be duplicating optimizations that the VHDL compiler would do, and the potential improvement in overall performance for a simulation is probably relatively small.

2. The primary purpose for manipulating the generated code during translation is to optimize it for synthesis, and simultaneously tailor it as much as possible for a specific synthesis system. At the current state-of-the-art in synthesis systems, certain types of VHDL constructs are "recognized" by the synthesizer, and will result either in more efficient hardware synthesis or a mapping onto a standard component in a library. The particular types of VHDL constructs which are most suitable also varies depending on the synthesis system. To the extent that the Translators could produce VHDL code well targeted to a specific synthesis system, and thus result in much more efficient hardware synthesized from that VHDL code, the potential benefits are very great. If this were done, then as another very interesting twist, we could also develop a VHDL-to-VHDL source code translator similar to the Ada and C Translators, which would have the effect of automatically preprocessing existing VHDL source code to target it to a specific synthesis environment.

Appendix A

Ada-to-VHDL Translator Example

This appendix shows an example Ada primitive from the RSS Q003 primitive set and the results of processing it with the Ada-to-VHDL Translator. The sections below show the original Ada source file, the VHDL source code produced by the Translator, and the automatically generated test driver program.

A.1 Ada Source Code

A.1.1 Package Specification

```
with COMMON_DATA_TYPES;
use  COMMON_DATA_TYPES;

package VOC_PWR_CFF is

--  NAME: VOC_PWR_CFF
--
--  TITLE: Complex Vector Power
--
--  CALLING SEQUENCE:
--
--  procedure Voc_Pwr
--   (N : in Pos_Int_Scalar;
--    X : in Cfloat_Types.Queue_of_Scalar;
--    Y : out Float_Types.Queue_of_Scalar;
--    Z : out Float_Scalar);
--
--  PARAMETERS:
--
--  N : Number of elements in X
--  X : Input vector
--  Y : Output vector
--  Z : Sum of elements
--
--  DESCRIPTION:
--
--  This primitive computes the magnitude squared of each element  of
--  a  complex  vector X, and sums the elements of the output vector,
--  Y.
--
--  RESTRICTIONS: None
--
--  OPTIONS: None
--
--  TIMING:
--
--  Mode(X) = CF         [NEB*(2*N+14) + 20]/7 usec
--  Mode(X) = DCF        [NEB*(2*N+17) + 26]/7 usec
--
--  DATA-DEPENDENT TIMING: None
--
--  ACCURACY: Single Precision Floating Point
--
--  SPACE REQUIREMENTS: Unknown
--
--  EXCEPTIONS: This primitive does not raise any specific exceptions
--
--  ALGORITHM:
--
--  /* Given an N-element complex input vector X*
--
--  Z = 0
--  FOR n = 1,N
--    Y(n) = X(n).re * X(n).re + X(n).im * X(n).im
--    Z = Z + Y(n)
--  END
--

  procedure VOC_PWR
     (N : in POS_INT_SCALAR;
```

```
      X : in CFLOAT_TYPES.QUEUE_OF_SCALAR;
      Y : out FLOAT_TYPES.QUEUE_OF_SCALAR;
      Z : out FLOAT_SCALAR);

  function  VOC_PWR_VALIDATE_PARAMETERS
     (N : in POS_INT_SCALAR;
      X : in CFLOAT_TYPES.QUEUE_OF_SCALAR) return INTEGER;

end VOC_PWR_CFF;
```

A.1.2 Package Body

```
package body VOC_PWR_CFF is

  subtype SQIT is SHELL_QUEUE_INDEX_TYPE;

  procedure VOC_PWR
       (N : in POS_INT_SCALAR;
        X : in CFLOAT_TYPES.QUEUE_OF_SCALAR;
        Y : out FLOAT_TYPES.QUEUE_OF_SCALAR;
        Z : out FLOAT_SCALAR) is

    SUM     : FLOAT_SCALAR := 0.0;
    Y_INDEX : SQIT := Y'FIRST;
    YTEMP   : FLOAT_SCALAR;

  begin

    for X_INDEX in X'RANGE loop
      YTEMP := X (X_INDEX).REAL** 2 + X (X_INDEX).IMAG** 2;
      SUM := SUM + YTEMP;
      Y (Y_INDEX) := YTEMP;
      Y_INDEX := Y_INDEX + 1;
    end loop;

    Z := SUM;

  end VOC_PWR;


  function  VOC_PWR_VALIDATE_PARAMETERS
       (N : in POS_INT_SCALAR;
        X : in CFLOAT_TYPES.QUEUE_OF_SCALAR) return INTEGER is

  begin

-- If all parameter values are valid, return 0.
-- If an individual value is invalid, return its parameter number.
-- If some combination of values are invalid, return a negative number.

-- This routine has no constraints or validation parameters required.

     return 0;

  end VOC_PWR_VALIDATE_PARAMETERS;

end VOC_PWR_CFF;
```

A.2 VHDL Source Code Produced by Translator

A.2.1 Package Specification

```
-- Source file is /jrs/hvt/test/suites/ada/prims/voc_pwr_cff_.ada
-- Source language is ADA
use work.COMMON_DATA_TYPES.all;
--  NAME: VOC_PWR_CFF
--
--  TITLE: Complex Vector Power
--
--  CALLING SEQUENCE:
--
--  procedure Voc_Pwr
--   (N : in Pos_Int_Scalar;
--    X : in Cfloat_Types.Queue_of_Scalar;
--    Y : out Float_Types.Queue_of_Scalar;
--    Z : out Float_Scalar);
--
--  PARAMETERS:
--
--  N : Number of elements in X
--  X : Input vector
--  Y : Output vector
--  Z : Sum of elements
--
--  DESCRIPTION:
--
--  This primitive computes the magnitude squared of each element  of
--  a  complex  vector X, and sums the elements of the output vector,
--  Y.
--
--  RESTRICTIONS: None
--
--  OPTIONS: None
--
--  TIMING:
--
--  Mode(X) = CF        [NEB*(2*N+14) + 20]/7 usec
--  Mode(X) = DCF       [NEB*(2*N+17) + 26]/7 usec
--
--  DATA-DEPENDENT TIMING: None
--
--  ACCURACY: Single Precision Floating Point
--
--  SPACE REQUIREMENTS: Unknown
--
--  EXCEPTIONS: This primitive does not raise any specific exceptions
--
--  ALGORITHM:
--
--  /* Given an N-element complex input vector X*
--
--  Z = 0
--  FOR n = 1,N
--    Y(n) = X(n).re * X(n).re + X(n).im * X(n).im
--    Z = Z + Y(n)
--  END
--
use work.HVT_Runtime_Library.all;
package VOC_PWR_CFF is
   procedure VOC_PWR
        (N : IN POS_INT_SCALAR;
         X : IN CFLOAT_TYPES_QUEUE_OF_SCALAR;
```

```
        Y : OUT FLOAT_TYPES_QUEUE_OF_SCALAR;
        Z : OUT FLOAT_SCALAR);
   procedure VOC_PWR_VALIDATE_PARAMETERS
        (N : IN POS_INT_SCALAR;
        X : IN CFLOAT_TYPES_QUEUE_OF_SCALAR;
        VOC_PWR_VALIDATE_PARAMETERS_Result : out INTEGER);
end VOC_PWR_CFF;
```

A.2.2 Package Body

```
-- Source file is /jrs/hvt/test/suites/ada/prims/voc_pwr_cff.ada
-- Source language is ADA
package body VOC_PWR_CFF is
   subtype SQIT is SHELL_QUEUE_INDEX_TYPE;
   procedure VOC_PWR
        (N : IN POS_INT_SCALAR;
         X : IN CFLOAT_TYPES_QUEUE_OF_SCALAR;
         Y : OUT FLOAT_TYPES_QUEUE_OF_SCALAR;
         Z : OUT FLOAT_SCALAR) is
      variable SUM : FLOAT_SCALAR := 0.0;
      variable Y_INDEX : SQIT := Y'Low;
      variable YTEMP : FLOAT_SCALAR;
   begin
      for X_INDEX in X'RANGE loop
         YTEMP := X(X_INDEX).REAL ** 2 + X(X_INDEX).IMAG ** 2;
         SUM := SUM + YTEMP;
         Y(Y_INDEX) := YTEMP;
         Y_INDEX := Y_INDEX + 1;
      end loop;
      Z := SUM;
   end VOC_PWR;

   procedure VOC_PWR_VALIDATE_PARAMETERS
        (N : IN POS_INT_SCALAR;
         X : IN CFLOAT_TYPES_QUEUE_OF_SCALAR;
         VOC_PWR_VALIDATE_PARAMETERS_Result : out INTEGER) is
   begin
      -- If all parameter values are valid, return 0.
      -- If an individual value is invalid, return its parameter number.
      -- If some combination of values are invalid, return a negative number.
      -- This routine has no constraints or validation parameters required.
      VOC_PWR_VALIDATE_PARAMETERS_Result := 0;
      return;
   end VOC_PWR_VALIDATE_PARAMETERS;

end VOC_PWR_CFF;
```

A.3 Generated Test Driver

```
-- Generated entity/architecture test driver program for program voc_pwr_cff.

use  Std.Textio.All;
use  Std.Simulator_Standard;
use  Work.Common_Data_Types.All;
use  Work.Hvt_Test_Support.All;

use Work.voc_pwr_cff.All;

entity voc_pwr_cff_Entity is
  generic (
           N_Val        : Integer;
           X_Dim_1      : Integer;
           X_File       : String;
           X_Offset     : Integer;
           Y_Dim_1      : Integer;
           Y_File       : String;
           Y_Offset     : Integer;
           Z_Val        : String;
           HVT_Debug    : Integer
          );

end voc_pwr_cff_Entity;


architecture voc_pwr_cff_Test of voc_pwr_cff_Entity is
begin
  process
    subtype Sqit is Shell_Queue_Index_Type;
    subtype Sfit is Shell_Family_Index_Type;

    variable Error_Count : Integer := 0;
    variable Output_Line : Line;

    -- Parameter declarations

    variable N : Int_Scalar := Int_Scalar(N_Val);
    variable X : CFloat_Types_Queue_of_Scalar (1 to Sqit(X_Dim_1));
    variable Y : Float_Types_Queue_of_Scalar (1 to Sqit(Y_Dim_1));
    variable Y_Correct : Float_Types_Queue_of_Scalar (1 to Sqit(Y_Dim_1));
    variable Z : Float_Scalar := Float_Value(Z_Val);
    variable Z_Correct : Float_Scalar := Float_Value(Z_Val);

  begin
    Simulator_Standard.Tracing_Off;

    -- Read datasets

    Read_Dataset (X_File, X_Offset, X);
    Read_Dataset (Y_File, Y_Offset, Y_Correct);

    if (HVT_Debug > 0) then
       write (output_line, string'("*** INPUT PARAMETER VALUES BEFORE PRIMITIVE EXECUTION ***"));
       writeline (output, output_line);
       write (output_line, string'(" "));
       writeline (output, output_line);
       Printout ("N", N);
       Printout ("X", X);
    end if;

    -- Call the program to be tested
```

```
    VOC_PWR (N, X, Y, Z);

    if (HVT_Debug > 0) then
       write (output_line, string'(" "));
       writeline (output, output_line);
       write (output_line, string'("*** OUTPUT PARAMETER VALUES AFTER PRIMITIVE EXECUTION ***"));
       writeline (output, output_line);
       write (output_line, string'(" "));
       writeline (output, output_line);
       Printout ("Y", Y);
       Printout ("Z", Z);
       write (output_line, string'(" "));
       writeline (output, output_line);
    end if;

    -- Compare output datasets to correct values.

    Compare ("Y", Y, Y_Correct, Error_Count);
    Compare ("Z", Z, Z_Correct, Error_Count);

    -- Check comparison results.

    if Error_Count = 0 then
       write (output_line, string'("Simulation results are CORRECT"));
       writeline (output, output_line);
    else
       write (output_line, string'("Simulation results are NOT CORRECT"));
       writeline (output, output_line);
    end if;

    write (output_line, string'(" "));
    writeline (output, output_line);
    Simulator_Standard.Terminate;
    Wait;

  end process;
end voc_pwr_cff_Test;
```

Appendix B

C-to-VHDL Translator Example

This appendix shows an example C primitive from the Mercury Software Algorithm Library (SAL) primitive set and the results of processing it with the C-to-VHDL Translator. The sections below show the original C source file, the VHDL source code produced by the Translator, and the automatically generated test driver program.

B.1 C Source Code

```
/*****************************************************************\
|***   MC Standard Algorithms -- "C" language Version          ***|
|*****************************************************************|
|*                                                               *|
|*       File Name:    CIMUL.MAC                                 *|
|*       Description:  Single Precision Complex Image Multiply   *|
|*       Entry/params: CIMUL (A, B, C, NR, NC, F)                *|
|*       Formula:   C(x,y) = A(x,y) * B(x,y)  if F = 1           *|
|*                  C(x,y) = conj of B(x,y) * A(x,y) if F = -1   *|
|*                                       for x=0 to NC-1         *|
|*                                       for y=0 to NR-1         *|
|*                                                               *|
|*           Mercury Computer Systems, Inc.                      *|
|*           Copyright (c) 1987  All rights reserved             *|
|*                                                               *|
|*  Revision      Date        Engineer; Reason                   *|
|*  --------      ----        ---------------                    *|
|*    0.0         870213       jvs;  Created                     *|
|*    0.1         900523       jg;   General stylistic revision  *|
|*    0.2         900615       jg;   Added Fortran Entry         *|
|*    0.3         900809       jg;   Removed "k"                 *|
\*****************************************************************/

#include "defs.h"

cimul_ (A, B, C, NR, NC, F)
float  *A;
float  *B;
float  *C;
N32    *NR;
N32    *NC;
I31    *F;

{
   cimul (A, B, C, *NR, *NC, *F);
}

cimul (A, B, C, NR, NC, F)
float  *A;
float  *B;
float  *C;
N32     NR;
N32     NC;
I31     F;

{
   float  ar, ai, br, bi, cr, ci, temp;
   N32    i, j, xoffset;

   xoffset = NC << 1;

   if (F >= 0) {
      for ( i = 0; i < 2; i++ ) {
         ar = *(A++);                  /* 1st two entry of 1st two rows */
         ai = *(A++);                  /* are special case {'Real' pairs} */
         br = *(B++);
         bi = *(B++);
         cr = ar * br;
         ci = ai * bi;
         *(C++) = cr;
         *(C++) = ci;                  /* Now rest of row is {Real,Image} */
```

```
      for ( j = 1; j < NC; j++ ) {
         ar = *(A++);
         ai = *(A++);
         br = *(B++);
         bi = *(B++);
         cr = ar * br;
         temp = ai * bi;
         cr = cr - temp;
         ci = ai * br;
         temp = ar * bi;
         ci = ci + temp;
         *(C++) = cr;
         *(C++) = ci;
      }
   }

   /* Now do rest of rows */

   for ( i = 2; i < NR; i++ ) {
      if ( !(i & 1) ) {
         for ( j = 0; j < 2; j++ ) {
            ar = *A;
            ai = *(A++ + xoffset);
            br = *B;
            bi = *(B++ + xoffset);
            cr = ar * br;
            temp = ai * bi;
            cr = cr - temp;
            ci = ai * br;
            temp = ar * bi;
            ci = ci + temp;
            *C = cr;
            *(C++ + xoffset) = ci;
         }
      }

      else {
         A += 2;
         B += 2;
         C += 2;
      }

      for ( j = 1; j < NC; j++ ) {
         ar = *(A++);
         ai = *(A++);
         br = *(B++);
         bi = *(B++);
         cr = ar * br;
         temp = ai * bi;
         cr = cr - temp;
         ci = ai * br;
         temp = ar * bi;
         ci = ci + temp;
         *(C++) = cr;
         *(C++) = ci;
      }
   }
}

else {                                  /* Conjugate */
   for ( i = 0; i < 2; i++ ) {
      ar = *(A++);                      /* 1st two entry of 1st two rows */
      ai = *(A++);                      /* are special case {'Real' pairs} */
      br = *(B++);
      bi = *(B++);
      cr = ar * br;
      ci = ai * bi;
```

```
            *(C++) = cr;
            *(C++) = ci;                    /* Now rest of row is {Real,Image} */

            for ( j = 1; j < NC; j++ ) {
               ar = *(A++);
               ai = *(A++);
               br = *(B++);
               bi = *(B++);
               cr = ar * br;
               temp = ai * bi;
               cr = cr + temp;
               ci = ai * br;
               temp = ar * bi;
               ci = ci - temp;
               *(C++) = cr;
               *(C++) = ci;
            }
         }

         /* Now do rest of rows */

         for ( i = 2; i < NR; i++ ) {
            if ( !(i & 1) ) {
               for ( j = 0; j < 2; j++ ) {
                  ar = *A;
                  ai = *(A++ + xoffset);
                  br = *B;
                  bi = *(B++ + xoffset);
                  cr = ar * br;
                  temp = ai * bi;
                  cr = cr + temp;
                  ci = ai * br;
                  temp = ar * bi;
                  ci = ci - temp;
                  *C = cr;
                  *(C++ + xoffset) = ci;
               }
            }

            else {
               A += 2;
               B += 2;
               C += 2;
            }

            for ( j = 1; j < NC; j++ ) {
               ar = *(A++);
               ai = *(A++);
               br = *(B++);
               bi = *(B++);
               cr = ar * br;
               temp = ai * bi;
               cr = cr + temp;
               ci = ai * br;
               temp = ar * bi;
               ci = ci - temp;
               *(C++) = cr;
               *(C++) = ci;
            }
         }
      }
   }
```

B.2 VHDL Source Code Produced by Translator

```
use work.C_Data_Types.all;
use work.HVT_Runtime_Library.all;
use work.defs.all;
use work.math.all;
use work.floatingpoint.all;
use work.ieeefp.all;
package m_cimul is
   -- Source file is temp215.c
   -- Source language is C
   --/*****************************************************************\
   --/*|***   MC Standard Algorithms -- "C" language Version         ***|
   --/*|*****************************************************************|
   --/*|*                                                               *|
   --/*|*        File Name:    CIMUL.MAC                                *|
   --/*|*        Description:  Single Precision Complex Image Multiply *|
   --/*|*        Entry/params: CIMUL (A, B, C, NR, NC, F)               *|
   --/*|*        Formula:   C(x,y) = A(x,y) * B(x,y)  if F = 1          *|
   --/*|*                   C(x,y) = conj of B(x,y) * A(x,y) if F = -1  *|
   --/*|*                                          for x=0 to NC-1      *|
   --/*|*                                          for y=0 to NR-1      *|
   --/*|*                                                               *|
   --/*|*            Mercury Computer Systems, Inc.                     *|
   --/*|*            Copyright (c) 1987  All rights reserved            *|
   --/*|*                                                               *|
   --/*|*  Revision      Date        Engineer; Reason                   *|
   --/*|*  --------      ----        ---------------                    *|
   --/*|*    0.0        870213        jvs;  Created                    *|
   --/*|*    0.1        900523        jg;   General stylistic revision *|
   --/*|*    0.2        900615        jg;   Added Fortran Entry        *|
   --/*|*    0.3        900809        jg;   Removed "k"                *|
   --/*\*****************************************************************/
   type N32_Array is array (Integer range <>) of N32;
   type I31_Array is array (Integer range <>) of I31;
   procedure cimulX
        (A_Array : inout Float_Array;
         B_Array : inout Float_Array;
         C_Array : inout Float_Array;
         NR_Array : inout N32_Array;
         NC_Array : inout N32_Array;
         F_Array : inout I31_Array;
         cimulX_Result : out Integer);
   procedure cimul
        (A_Array : inout Float_Array;
         B_Array : inout Float_Array;
         C_Array : inout Float_Array;
         NR : in N32;
         NC : in N32;
         F : in I31;
         cimul_Result : out Integer);
end m_cimul;

package body m_cimul is
   type TEXT_Array is array (Integer range <>) of TEXT;
   type N16_Array is array (Integer range <>) of N16;
   type sigfpe_handler_type_Array is array (Integer range <>) of sigfpe_handler_type;
   procedure cimulX
        (A_Array : inout Float_Array;
         B_Array : inout Float_Array;
         C_Array : inout Float_Array;
         NR_Array : inout N32_Array;
         NC_Array : inout N32_Array;
         F_Array : inout I31_Array;
         cimulX_Result : out Integer) is
```

```
      variable A : Pointer_to_Float := A_Array'Low;
      variable B : Pointer_to_Float := B_Array'Low;
      variable C : Pointer_to_Float := C_Array'Low;
      variable NR : Pointer := NR_Array'Low;
      variable NC : Pointer := NC_Array'Low;
      variable F : Pointer := F_Array'Low;
      variable cimul_1 : Integer;
      variable fp_direction : fp_direction_type;                --*GLOBAL*-- variable
      variable fp_precision : fp_precision_type;                --*GLOBAL*-- variable
      variable fp_accrued_exceptions : fp_exception_field_type;         --*GLOBAL*-- variable
      variable ieee_handlers : sigfpe_handler_type_Array (0 to 5);              --*GLOBAL*-
- variable
      variable errno : Integer;          --*GLOBAL*-- variable
      variable fp_pi : fp_pi_type;               --*GLOBAL*-- variable
      variable signgam : Integer;                --*GLOBAL*-- variable
   begin
      cimul (A_Array, B_Array, C_Array, NR_Array(NR), NC_Array(NC), F_Array(F), cimul_1);
   end cimulX;

   procedure cimul
        (A_Array : inout Float_Array;
         B_Array : inout Float_Array;
         C_Array : inout Float_Array;
         NR : in N32;
         NC : in N32;
         F : in I31;
         cimul_Result : out Integer) is
      variable A : Pointer_to_Float := A_Array'Low;
      variable B : Pointer_to_Float := B_Array'Low;
      variable C : Pointer_to_Float := C_Array'Low;
      variable fp_direction : fp_direction_type;                --*GLOBAL*-- variable
      variable fp_precision : fp_precision_type;                --*GLOBAL*-- variable
      variable fp_accrued_exceptions : fp_exception_field_type;         --*GLOBAL*-- variable
      variable ieee_handlers : sigfpe_handler_type_Array (0 to 5);              --*GLOBAL*-
- variable
      variable errno : Integer;          --*GLOBAL*-- variable
      variable fp_pi : fp_pi_type;               --*GLOBAL*-- variable
      variable signgam : Integer;                --*GLOBAL*-- variable
      variable ar : Real;
      variable ai : Real;
      variable br : Real;
      variable bi : Real;
      variable cr : Real;
      variable ci : Real;
      variable temp : Real;
      variable i : N32;
      variable j : N32;
      variable xoffset : N32;
   begin
      xoffset := lib_shl(NC, 1);
      if F >= 0 then
         i := 0;
         while i < 2 loop
            ar := A_Array(A);
            A := A + 1;
            --/* 1st two entry of 1st two rows */
            ai := A_Array(A);
            A := A + 1;
            --/* are special case {'Real' pairs} */
            br := B_Array(B);
            B := B + 1;
            bi := B_Array(B);
            B := B + 1;
            cr := ar * br;
            ci := ai * bi;
            C_Array(C) := cr;
            C := C + 1;
```

```
   C_Array(C) := ci;
   C := C + 1;
   --/* Now rest of row is {Real,Image} */
   j := 1;
   while j < NC loop
      ar := A_Array(A);
      A := A + 1;
      ai := A_Array(A);
      A := A + 1;
      br := B_Array(B);
      B := B + 1;
      bi := B_Array(B);
      B := B + 1;
      cr := ar * br;
      temp := ai * bi;
      cr := cr - temp;
      ci := ai * br;
      temp := ar * bi;
      ci := ci + temp;
      C_Array(C) := cr;
      C := C + 1;
      C_Array(C) := ci;
      C := C + 1;
      j := j + 1;
   end loop;
   i := i + 1;
end loop;
--/* Now do rest of rows */
i := 2;
while i < NR loop
   if (lib_and(i, 1)) = 0 then
      j := 0;
      while j < 2 loop
         ar := A_Array(A);
         ai := A_Array(A + xoffset);
         A := A + 1;
         br := B_Array(B);
         bi := B_Array(B + xoffset);
         B := B + 1;
         cr := ar * br;
         temp := ai * bi;
         cr := cr - temp;
         ci := ai * br;
         temp := ar * bi;
         ci := ci + temp;
         C_Array(C) := cr;
         C_Array(C + xoffset) := ci;
         C := C + 1;
         j := j + 1;
      end loop;
   else
      A := A + 2;
      B := B + 2;
      C := C + 2;
   end if;
   j := 1;
   while j < NC loop
      ar := A_Array(A);
      A := A + 1;
      ai := A_Array(A);
      A := A + 1;
      br := B_Array(B);
      B := B + 1;
      bi := B_Array(B);
      B := B + 1;
      cr := ar * br;
      temp := ai * bi;
```

```
            cr := cr - temp;
            ci := ai * br;
            temp := ar * bi;
            ci := ci + temp;
            C_Array(C) := cr;
            C := C + 1;
            C_Array(C) := ci;
            C := C + 1;
            j := j + 1;
         end loop;
         i := i + 1;
      end loop;
   else
      --/* Conjugate */
      i := 0;
      while i < 2 loop
         ar := A_Array(A);
         A := A + 1;
         --/* 1st two entry of 1st two rows */
         ai := A_Array(A);
         A := A + 1;
         --/* are special case {'Real' pairs} */
         br := B_Array(B);
         B := B + 1;
         bi := B_Array(B);
         B := B + 1;
         cr := ar * br;
         ci := ai * bi;
         C_Array(C) := cr;
         C := C + 1;
         C_Array(C) := ci;
         C := C + 1;
         --/* Now rest of row is {Real,Image} */
         j := 1;
         while j < NC loop
            ar := A_Array(A);
            A := A + 1;
            ai := A_Array(A);
            A := A + 1;
            br := B_Array(B);
            B := B + 1;
            bi := B_Array(B);
            B := B + 1;
            cr := ar * br;
            temp := ai * bi;
            cr := cr + temp;
            ci := ai * br;
            temp := ar * bi;
            ci := ci - temp;
            C_Array(C) := cr;
            C := C + 1;
            C_Array(C) := ci;
            C := C + 1;
            j := j + 1;
         end loop;
         i := i + 1;
      end loop;
      --/* Now do rest of rows */
      i := 2;
      while i < NR loop
         if (lib_and(i, 1)) = 0 then
            j := 0;
            while j < 2 loop
               ar := A_Array(A);
               ai := A_Array(A + xoffset);
               A := A + 1;
               br := B_Array(B);
```

```
                bi := B_Array(B + xoffset);
                B := B + 1;
                cr := ar * br;
                temp := ai * bi;
                cr := cr + temp;
                ci := ai * br;
                temp := ar * bi;
                ci := ci - temp;
                C_Array(C) := cr;
                C_Array(C + xoffset) := ci;
                C := C + 1;
                j := j + 1;
             end loop;
          else
             A := A + 2;
             B := B + 2;
             C := C + 2;
          end if;
          j := 1;
          while j < NC loop
             ar := A_Array(A);
             A := A + 1;
             ai := A_Array(A);
             A := A + 1;
             br := B_Array(B);
             B := B + 1;
             bi := B_Array(B);
             B := B + 1;
             cr := ar * br;
             temp := ai * bi;
             cr := cr + temp;
             ci := ai * br;
             temp := ar * bi;
             ci := ci - temp;
             C_Array(C) := cr;
             C := C + 1;
             C_Array(C) := ci;
             C := C + 1;
             j := j + 1;
          end loop;
          i := i + 1;
       end loop;
    end if;
 end cimul;

end m_cimul;
```

B.3 Generated Test Driver

```
-- Generated entity/architecture test driver program for program m_cimul.

use  Std.Textio.All;
use  Std.Simulator_Standard;
use  Work.C_Data_Types.All;
use  Work.Hvt_Test_Support.All;

use Work.m_cimul.All;

entity m_cimul_Entity is
  generic (
           A_Array_Dim_1        : Integer;
           A_Array_File : String;
           A_Array_Offset       : Integer;
           B_Array_Dim_1        : Integer;
           B_Array_File : String;
           B_Array_Offset       : Integer;
           C_Array_Dim_1        : Integer;
           C_Array_File : String;
           C_Array_Offset       : Integer;
           NR_Val       : Integer;
           NC_Val       : Integer;
           F_Val        : Integer;
           cimul_Result_Val     : Integer;
           HVT_Debug    : Integer
          );

end m_cimul_Entity;


architecture m_cimul_Test of m_cimul_Entity is
begin
  process
    variable Error_Count : Integer := 0;
    variable Output_Line : Line;

    -- Parameter declarations

    variable A_Array : Float_Array (0 to A_Array_Dim_1 - 1);
    variable A_Array_Correct : Float_Array (0 to A_Array_Dim_1 - 1);
    variable B_Array : Float_Array (0 to B_Array_Dim_1 - 1);
    variable B_Array_Correct : Float_Array (0 to B_Array_Dim_1 - 1);
    variable C_Array : Float_Array (0 to C_Array_Dim_1 - 1);
    variable C_Array_Correct : Float_Array (0 to C_Array_Dim_1 - 1);
    variable NR : Integer := NR_Val;
    variable NC : Integer := NC_Val;
    variable F : Integer := F_Val;
    variable cimul_Result : Integer;
    variable cimul_Result_Correct : Integer := cimul_Result_Val;

  begin
    Simulator_Standard.Tracing_Off;

    -- Read datasets

    Read_Dataset (A_Array_File, A_Array_Offset, A_Array);
    A_Array_Correct := A_Array;
    Read_Dataset (B_Array_File, B_Array_Offset, B_Array);
    B_Array_Correct := B_Array;
    Read_Dataset (C_Array_File, C_Array_Offset, C_Array);
    C_Array_Correct := C_Array;
```

```
    if (HVT_Debug > 0) then
       write (output_line, string'("*** INPUT PARAMETER VALUES BEFORE PRIMITIVE EXECUTION ***"));
       writeline (output, output_line);
       write (output_line, string'(" "));
       writeline (output, output_line);
       Printout ("A_Array", A_Array);
       Printout ("B_Array", B_Array);
       Printout ("C_Array", C_Array);
       Printout ("NR", NR);
       Printout ("NC", NC);
       Printout ("F", F);
    end if;

    -- Call the program to be tested

    cimul (A_Array, B_Array, C_Array, NR, NC, F, cimul_Result);

    if (HVT_Debug > 0) then
       write (output_line, string'(" "));
       writeline (output, output_line);
       write (output_line, string'("*** OUTPUT PARAMETER VALUES AFTER PRIMITIVE EXECUTION ***"));
       writeline (output, output_line);
       write (output_line, string'(" "));
       writeline (output, output_line);
       Printout ("A_Array", A_Array);
       Printout ("B_Array", B_Array);
       Printout ("C_Array", C_Array);
       Printout ("cimul_Result", cimul_Result);
       write (output_line, string'(" "));
       writeline (output, output_line);
    end if;

    -- Compare output datasets to correct values.

    Compare ("A_Array", A_Array, A_Array_Correct, Error_Count);
    Compare ("B_Array", B_Array, B_Array_Correct, Error_Count);
    Compare ("C_Array", C_Array, C_Array_Correct, Error_Count);
    Compare ("cimul_Result", cimul_Result, cimul_Result_Correct, Error_Count);

    -- Check comparison results.

    if Error_Count = 0 then
       write (output_line, string'("Simulation results are CORRECT"));
       writeline (output, output_line);
    else
       write (output_line, string'("Simulation results are NOT CORRECT"));
       writeline (output, output_line);
    end if;

    write (output_line, string'(" "));
    writeline (output, output_line);
    Simulator_Standard.Terminate;
    Wait;

  end process;
end m_cimul_Test;
```